Цифровая схемотехника и архитектура компьютера: RISC-V

Дэвид М. Харрис
Сара Л. Харрис

ВЫСШАЯ · ШКОЛА · ЭКОНОМИКИ ·
МИЭМ
ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Сара Л. Харрис
Дэвид Харрис

ЦИФРОВАЯ СХЕМОТЕХНИКА И АРХИТЕКТУРА КОМПЬЮТЕРА: RISC-V

Под редакцией А. Ю. Романова

Digital Design and Computer Architecture

RISC-V Edition

Sarah L. Harris
David Harris

AMSTERDAM · BOSTON · HEIDELBERG · LONDON
NEW YORK · OXFORD · PARIS · SAN DIEGO
SAN FRANCISCO · SINGAPORE · SYDNEY · TOKYO
Morgan Kaufmann is an imprint of Elsevier

Цифровая схемотехника и архитектура компьютера: RISC-V

Сара Л. Харрис
Дэвид Харрис

Под редакцией А. Ю. Романова

ДМК ПРЕСС издательство
Москва, 2022

УДК 004.2+744.4
ББК 32.971.3
Х21

Научный редактор:
Романов А. Ю., канд. тех. наук,
доцент Московского института электроники и математики им. А. Н. Тихонова
Национального исследовательского университета «Высшая школа экономики»

Сара Л. Харрис, Дэвид Харрис

Х21 Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с.: ил.

ISBN 978-5-97060-961-3

В книге представлен уникальный и современный подход к разработке цифровых устройств. Авторы начинают с цифровых логических элементов, переходят к разработке комбинационных и последовательностных схем, а затем используют эти базовые блоки как основу для самого сложного: разработки настоящего процессора RISC-V. По всему тексту приводятся примеры на языках SystemVerilog и VHDL, иллюстрирующие методы и способы разработки схем с помощью САПР. Изучив эту книгу, читатели смогут разработать свой собственный микропроцессор и получат полное понимание того, как он работает. В книге объединен привлекательный и юмористический стиль изложения с развитым и практичным подходом к разработке цифровых устройств.

В издание вошли новые материалы о системах ввода/вывода применительно к процессорам общего назначения как для ПК, так и для микроконтроллеров. Приведены практические примеры интерфейсов периферийных устройств с применением RS-232, SPI, управления двигателями, прерываний, беспроводной связи и аналого-цифрового преобразования. Представлено высокоуровневое описание интерфейсов, включая USB, SDRAM, Wi-Fi, PCI Express и др.

Издание будет полезно студентам, инженерам, а также широкому кругу читателей, интересующихся современной схемотехникой.

ISBN 978-0-12-820064-3 (англ.)
ISBN 978-5-97060-961-3 (рус.)

Содержание

Отзывы на книгу «Цифровая схемотехника и архитектура компьютера. RISC-V»

Харрис и Харрис детально описали устройство процессора RISC-V от электронных компонентов до микроархитектуры. Их ясные объяснения в сочетании с широким охватом темы дают полное представление как о цифровой схемотехнике, так и об архитектуре RISC-V. Это очень информативный и познавательный подход, поскольку у студентов есть отличная возможность запускать большие цифровые проекты на современных FPGA.

Дэвид А. Паттерсон, Калифорнийский университет в Беркли

Потрясающе, какие разнообразные знания авторы объединили в одной книге! По мере развития производства полупроводников значимость правильной разработки цифровых схем и компьютерной архитектуры будет только возрастать. Читатели найдут доступное и всестороннее рассмотрение обеих тем и после прочтения книги получат четкое понимание архитектуры набора команд RISC-V.

Эндрю Уотерман, SiFive

Мне доводилось видеть отличные учебники по цифровой схемотехнике и отличные учебники по компьютерным архитектурам – но этот учебник представляет собой и то, и другое! Он также уникален своей способностью формировать общую картину. Авторы начинают с азов, и это делает архитектуру RISC-V понятной. Упражнения к главам этой книги послужат отличным методическим ресурсом для университетских преподавателей.

Рой Кравиц, Государственный университет Портленда

Когда я впервые прочитал учебник по MIPS в 2008 году, то подумал, что это один из лучших учебников по компьютерной архитектуре. Я сразу начал использовать его в своих лекциях. Тринадцать лет спустя мне посчастливилось прочитать новое издание про RISC-V, и мое мнение осталось прежним: это отличная книга, очень понятная, исчерпывающая, с высоким образовательным потенциалом. Она полностью соответствует учебной программе, которую проходят студенты в области цифровой схемотехники и компьютерной архитектуры. Я с нетерпением жду возможности использовать этот учебник по архитектуре RISC-V в своих лекциях.

Даниэль Чавер Мартинес, Мадридский университет Комплутенсе

Об авторах

Дэвид Мани Харрис (David Money Harris) – доцент в колледже им. Харви Мадда (Harvey Mudd College). Получил ученую степень кандидата наук по электронике в Стэнфордском университете и степень магистра по электронике и информатике в Массачусетском технологическом институте (MIT). Перед Стэнфордом работал в компании Intel в качестве схемотехника и разработчика логики для процессоров Itanium и Pentium II. Впоследствии работал консультантом в Sun Microsystems, Hewlett-Packard, Evans & Sutherland и других компаниях.

Увлечения Дэвида включают в себя преподавание, разработку чипов и активный отдых на природе. В свободное от работы время занимается пешим туризмом, скалолазанием и альпинизмом. Особенно любит длинные прогулки с сыном Абрахамом, который родился, когда Дэвид начал работать над этой книгой. Дэвид имеет более десяти патентов и является автором трех других учебников по разработке чипов, а также двух путеводителей по горам Южной Калифорнии.

Сара Л. Харрис (Sarah L. Harris) – доцент в колледже им. Харви Мадда (Harvey Mudd College). Получила степени магистра и кандидата наук по электронике в Стэнфордском университете и степень бакалавра по электронике и вычислительной технике в университете Брайама Янга (BrighamYoung University). Сара также работала в компаниях Hewlett-Packard, San Diego Supercomputer Center, Nvidia и исследовательском отделе компании Microsoft Research в Пекине.

Интересы Сары не ограничиваются преподаванием, изучением и разработкой новых технологий, она также любит путешествовать, увлекается виндсерфингом, скалолазанием и игрой на гитаре. Среди ее недавних начинаний можно отметить исследования в области интерфейсов, позволяющих разрабатывать цифровые электрические схемы простыми рисунками от руки, работу в качестве научного корреспондента для филиала Национального общественного радио (National Public Radio) и обучение кайтсерфингу. Сара говорит на четырех языках и собирается изучить еще несколько в ближайшем будущем.

Романов Александр Юрьевич – научный редактор русского перевода данной книги, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ). В 2009 г. закончил магистратуру в Харьковском политехническом институте, работал в Киевском политехническом институте им. Сикорского. С 2014 г. работает в МИЭМ НИУ ВШЭ, где возглавляет лабораторию САПР (https://miem.hse.ru/edu/ce/cadsystem), специализирующуюся на проектной деятельности, а также разработке цифровых систем на ПЛИС/микроконтроллерах, робототехнических комплексов, аппаратных реализаций систем искусственного интеллекта, многопроцессорных систем, систем удаленного доступа к лабораторному оборудованию и т. д. В 2015 г. защитил диссертацию в Институте проблем проектирования в микроэлектронике РАН (г. Зеленоград), является автором более 150 научных статей, патентов и книг. Более подробно об учебном процессе в лаборатории можно узнать из интервью: https://miem.hse.ru/news/364316102.html.

Предисловие к русскому изданию

Вы держите в руках книгу, которая занимает на российском книжном рынке особое место. Если вы студент и хотите пройти собеседование в крупную электронную компанию на прозицию проектировщика процессоров, нейроускорителей или сетевых микросхем, то самое лучшее, что вы можете сделать сейчас, – это прочитать данную книгу от корки до корки, одновременно выполняя упражнения на симуляторах и платах ПЛИС.

Когда мы говорим о собеседованиях, мы говорим о таких компаниях, как Apple, Intel, NVidia, а также о передовых российских проектировщиках процессоров Syntacore, «Элвис-НеоТек» и «Байкал Электроникс». В каждой из них вам дадут задания типа «напишите на доске дизайн простого арбитра на языке описания аппаратуры Verilog» или «объясните, как помогают производительности микропроцессора байпасы в его конвейере».

Конечно, мы не утверждаем, что изучение этого учебника гарантирует вам успех, но эта книга закладывает современную базу во всех областях, о которых вас будут спрашивать: цифровая логика и ее тайминг, арифметические блоки и конечные автоматы, архитектура (система команд) и микроархитектура (строение конвейера) процессора. С использованием того же самого языка SystemVerilog, который используют современные разработчики цифровых систем на рабочем месте (вам также могут встретиться блоки на языке VHDL, и он тоже есть в книге).

После этой книги вам, конечно, нужно будет сделать несколько учебных проектов и изучить по статьям в сети Интернет некоторые типы дизайнов, которых в книге нет (очереди FIFO, пересечение доменов тактовой частоты и т. д.). Совместно с этой книгой также рекомендуется читать еще одну – «Цифровой синтез: практический курс»[1]. Она создана специально как дополнение к предыдущей версии книги Харрисов; в ближайшее время планируется ее переиздание, адаптированное под RISC-V. После этого вы будете готовы к бою. Никакая другая книга или комбинация книг на русском языке не поможет вам пройти эту начальную часть траектории эффективнее, чем «Цифровая схемотехника и архитектура компьютера: RISC-V» Дэвида Харриса и Сары Харрис.

Как возникла современная база проектирования

В 1980-е годы произошли две революции в проектировании цифровых микросхем. Первая революция была в маршруте проектирования. До конца 1980-х схемы рисо-

[1] Цифровой синтез: практический курс / под общ. ред. А. Ю. Романова, Ю. В. Панчула. М.: ДМК Пресс, 2020. https://dmkpress.com/catalog/electronics/circuit_design/978-5-97060-850-0/.

вали мышкой на экране, а с начала 1990-х их стали синтезировать из кода на языках описания аппаратуры Verilog и VHDL. Основные события:

- 1984 – Gateway Design Automation / Cadence изобретают язык описания аппаратуры Verilog;
- 1984 – Xilinx изобретает реконфигурируемые микросхемы ПЛИС/FPGA;
- 1986 – Optimal Solutions / Synopsys изобретают цифровой синтез;
- 1988–1992 – цифровой синтез внедряют в проектирование Apple, Sun, Nokia и др.;
- 1997–1999 – Lexra, MIPS, ARM начинают лицензировать процессорные ядра в виде IP-блоков (Intellectual Property – интеллектуальная собственность).

Вторая революция произошла в архитектуре и микроархитектуре процессоров. В 1970-х были популярны процессоры с двухуровневой организацией на основе так называемой технологии микропрограммирования. Команды процессора, видимые программисту, реализовывались на аппаратном уровне с помощью цепочек из слов (последовательностей битов в памяти) с сигналами контроля, так называемого микрокода. Такая организация позволяла создавать очень сложные системы команд, но ограничивала возможности по их параллельному выполнению.

В 1978 году группа исследователей в Стенфорде под руководством Джона Хеннесси задала себе вопрос: действительно ли нужны эти сложные команды, или их необходимость – просто маркетинговая иллюзия? Стенфордцы провели анализ большого количества пользовательских программ и пришли к выводу, что большинство используемых в программах команд – простые. И если тратить усилия не на усложнение цепочек микрокода, а на построение так называемого конвейера – структуры, в которой простые команды выполняются с перекрытием во времени, – то можно выполнять программы быстрее. Так появилась архитектура MIPS.

К похожим идеям пришла группа в Беркли под руководством Дэвида Паттерсона, которая в начале 1980-х создала архитектуры RISC I и RISC II, из которых выросла архитектура SPARC. В середине 1980-х появилась компания ARM, и за последующие десятилетия процессоры с новой организацией сначала завоевали рынок рабочих станций, а потом и бытовой электроники, сотовых телефонов и микроконтроллеров.

В конце 1980-х даже Intel, которая изначально делала процессоры на основе микрокода, стала вводить в Intel 486 конвейер, а к 1996 году построила процессор PentiumPro, в котором большинство команд на лету преобразовывались в простые команды, отправлявшиеся на конвейер в стиле RISC-процессоров. Хранимый в памяти микрокод остался только для сложных инструкций.

В начале 1990-х основатели концепции RISC-процессоров Джон Хеннесси и Дэвид Паттерсон опубликовали два учебника, которые стали бестеллерами:

- учебник начального уровня «Архитектура компьютера и проектирование компьютерных систем»
- и более сложный учебник «Компьютерная архитектура: количественный подход».

Эти учебники описывали архитектуру и микроархитектуру сначала на основе MIPS-образной архитектуры DLX, а потом стали использовать MIPS. К тому времени процессоры архитектуры MIPS уже использовались в компьютерах для голливудских спецэффектов, а потом и в домашней электронике.

В течение 1990-х американские университеты внедрили в учебный процесс книги Хеннесси и Паттерсона, курсы по языкам описания аппаратуры Verilog и VHDL, а также лабораторные работы на платах с микросхемами реконфигурируемой логики ПЛИС/FPGA, которые позволили строить студенческие процессоры без сложной процедуры заказа их на фабрике. Так выросло поколение студентов, которые разработали Apple iPhone, графические процессоры от NVidia, микросхемы для маршрутизаторов Cisco и Juniper и другие популярные устройства.

Что происходило в это время в России

Революции в цифровом синтезе и микроархитектуре процессоров по времени выпали на сложный период российской истории. Открытие советского рынка для иностранных компьютеров, коллапс СССР и недофинансирование вузов привели к тому, что в российском обществе перестали верить, что в России возможно проектирование конкурентоспособных чипов.

Долгое время группы разработчиков сохранялись только в компаниях, связанных с обороной и космосом, для проектирования чипов для космоса в таких организациях, как НИИСИ и НПЦ «Элвис». Российская команда, разработавшая процессор «Эльбрус», прототип которого при симуляции на Verilog показывал многообещающие результаты на вычислениях с плавающей запятой, попыталась в 2000 году получить финансирование у венчурных капиталистов Кремниевой долины, но вернулась в Россию.

В результате обучение компьютерной архитектуре во многих российских вузах стало описательным. Например, вузовские преподаватели стали использовать учебник Эндрю Таненбаума «Архитектура компьютеров», который был больше ориентирован на программистов, чем разработчиков процессоров. Что и понятно – Таненбаум получил известность как создатель операционной системы Minix, предшественницы Linux, а не разработчик процессора. Для микроархитектуры учебник использовал предыдущую технологическую базу (микрокод) и никак не был привязан к синтезу процессоров на языках описания аппаратуры. То есть студенты изучали системы команд и виды кеша для программистов, но не могли сделать процессор руками.

Учебники Паттерсона и Хеннесси были переведены на русский язык с большим опозданием, и в них не вошли приложения с описанием языков проектирования аппаратуры. Профессор Аркадий Поляков после работы в Кремниевой долине вернулся в Россию и издал в 2003 году учебник по Verilog, но в нем не было привязки к компьютерной архитектуре. Даже когда российские вузы делали лабораторные работы с ПЛИС, преподаватели часто выбирали разработку схемы с помощью рисования мышкой на экране, хотя в американских компаниях это перестали делать еще в начале 1990-х. В типичной вузовской методичке по цифровой электронике 2000-х годов шло качественное описание схем мультиплексоров и триггеров, а потом, пропустив

два уровня абстракции, студенты сразу изучали программирование микроконтроллеров. Не было учебника, который бы увязывал все эти элементы в одно целое.

История появления учебника «Цифровая схемотехника и архитектура компьютера»

Дэвид Харрис учился в MIT как раз тогда, когда произошла революция в маршруте проектирования конца 1980 – начала 1990-х годов. Вооруженный новыми методологиями, Дэвид пошел работать в Intel над процессором Pentium II. После этого защитил диссертацию в Стенфорде и стал преподавателем в Колледже Харви-Мадд в южной Калифорнии.

- http://pages.hmc.edu/harris/about/General_Resume.pdf.

Колледж Харви-Мадд не особо известен широкой публике, но находится среди топ-университетов по заработным платам выпускников, а также количеству выпускников, защищающих впоследствии диссертации. Еще Харви-Мадд известен проектами в области робототехники, которые они делают вместе с NASA. Иными словами, это практик высшего калибра.

- https://www.monster.com/career-advice/article/colleges-that-get-most-pay-for-graduates.
- https://www.hmc.edu/about-hmc/2020/09/14/harvey-mudd-ranks-high-in-u-s-news-and-world-report-2021/.
- https://ti.arc.nasa.gov/news/ASR-hosts-Clinic-project/.

Дэвид Харрис и его коллега Сара Харрис (они не родственники, а просто однофамильцы) в 2008 году написали первый вариант учебника, в котором в лаконичной и технически корректной форме изложили материал, который обычно входил в несколько учебников: цифровая логика, языки описания аппаратуры Verilog и VHDL, архитектура и микроархитектура компьютера, а также использование готовых чипов. Студенты получили возможность, используя только один учебник, начать с нуля, дойти до конструирования собственного небольшого процессора, реализующего подмножество архитектуры MIPS, а потом сравнить его работу с реальным микроконтроллером Microchip PIC32 на архитектуре MIPS.

Книга Харрисов появилась в России

В начале 2010-х годов в российской электронной индустрии наступило оживление. Зеленоградские компании «Элвис» и «Миландр» налаживали контакты с ARM и MIPS для лицензирования процессорных ядер, НИИСИ строил суперскалярное 64-битное MIPS-ядро, КМ211 разрабатывали процессоры для смарт-карт и налаживали контакты с тайваньской фабрикой TSMC. РОСНАНО финансировало проект компании «Элвис» в области умных камер и новую компанию «Байкал Электроникс».

Когда проблемы недостатка финансирования и изоляции российских компаний от международного рынка стали решаться, на первый план вышла проблема нехватки кадров. Хотя вузовские программы в МИЭТ и ИТМО старались поддерживать свои программы на уровне, компаниям приходилось обучать не только разработчиков схем на Verilog (на уровне RTL – Register Transfer Level), но и инженеров-верификаторов, которым нужно было создавать тесты и модели со знанием, что происходит в схеме, спроектированной на Verilog.

Поэтому когда в 2014 году появилась идея перевести на русский язык книгу Харрисов, ее поддержали сразу несколько человек и компаний. Преподаватели и аспиранты российских университетов МИФИ, ИТМО, ИТМиВТ, СПб ГУАП, украинских КНУ, КПИ, ХНУРЭ и ЧНТУ; сотрудники российских компаний МЦСТ, НИИСИ РАН, «Модуль», RusBITech, amperka.ru, Runtime Design Automation, «БиДжи»; русские инженеры американских и европейских компаний Imagination Technologies / MIPS Processors, AMD, Synopsys, Apple, eASIC, Cadence, NVidia, Marvell Semiconductor, университета Принстон – более 40 человек приняли участие в переводе, ревью, редактировании и корректировании как учебника, так и лекционных слайдов для него.

Перевод поддержала британская компания Imagination Technologies, которая в это время заключала сделки по лицензированию процессорных ядер MIPS и графических ядер PowerVR с российскими компаниями и была заинтересована в улучшении технического образования в России для налаживания долговременных бизнес-отношений с российскими партнерами. В издании книги также помогло eNano, образовательное отделение РОСНАНО, российского фонда, который вкладывал в микроэлектронные проекты.

После выхода первого онлайн-издания за дело взялось российское издательство «ДМК Пресс», которое выпустило второе издание Харрисов (использующее архитектуру MIPS) в бумажном виде, затем дополнение, которое применяет архитектуру ARM. Книга «Цифровая схемотехника и архитектура компьютера» стала настолько популярна, что ее начали использовать в ведущих российских вузах. Единственного, чего ей не хватало, это полноценного практического курса, который бы мог дополнить основной материал лабораторными работами. В 2019 г. такой курс был создан. Под эгидой МИЭМ НИУ ВШЭ была собрана большая команда преподавателей и разработчиков из СНГ и США, написавшие книгу «Цифровой синтез: практический курс» под редакцией А. Ю. Романова и Ю. В. Панчула. Книга хороша тем, что она раскрывает и дополняет материал книги Харрисов, а также поддержана репозитарием с исходными кодами всех примеров, приведенных в ней, и адаптирована под выполнение лабораторных работ на дешевых отладочных платах с ПЛИС.

И вот, наконец, ввиду все большего распространения архитектуры RISC-V, появилось новое издание книги «Цифровая схемотехника и архитектура компьютера».

Почему RISC-V?

Лицензируемые ядра RISC-процессоров совершили еще одну революцию в конце 1990 – начале 2000-х годов, когда ARM стал сердцем сотовых телефонов от Nokia и Ericsson, а MIPS стали использовать в телевизорах Sony, игровых приставках

и даже роботах. К компаниям ARM и MIPS присоединились несколько конкурентов, в частности ARC и Tensilica, которые образовали так называемую индустрию полупроводниковой интеллектуальной собственности, semiconductor IP, общим размеров в несколько миллиардов долларов.

Помимо разработчиков центральных процессоров в эту индустрию вошли Imagination Technologies – компания, которая спроектировала графический процессор PowerVR для ранних Apple iPhone, затем разработчик процессора для обработки сигналов CEVA и уже в наше время компании, которые выпускают ускорители нейросетевых вычислений.

ARM и MIPS получали доход двумя способами:

1) продажей лицензий на процессорные ядра – фактически на использование сотни тысяч строк на Verilog, написанных инженерами ARM и MIPS, внутри систем на кристалле заказчика. Примерами таких компаний стали Microchip, которая лицензировала ядро MIPS M4K для микроконтроллеров PIC32, и ST Microelectronics, которая лицензировала ядра ARM Cortex M для линейки микроконтроллеров STM32;
2) продажей так называемой архитектурной лицензии – права на создание процессора собственной микроархитектуры. Инженеры компании-покупателя архитектурной лицензии создавали собственную микроархитектуру и могли разрабатывать код на Verilog сами, но их ядро делалось совместимым по архитектуре (системе команд) с ARM или MIPS. Последним примером такого лицензиата является компания Apple, которая создала свое ARM-совместимое ядро для системы на кристалле Apple M1.

Хотя разделение компаний на разработчиков IP-блоков и разработчиков систем на кристалле помогло развить индустрию в 1990–2000-е годы, не все в этой схеме было идеальным.

- Во-первых, многие компании были недовольны условиями и политикой лицензирования как ядер, так и архитектуры. Особенно сильное негативное впечатление на индустрию произвел судебный процесс MIPS против Lexra в 1999 году, в результате которого пионер IP-лицензирования компания Lexra обанкротилась из-за довольно мелкого нарушения патента на редко используемые инструкции невыравненного обмена с памятью (https://www.eetimes.com/lexra-quits-ip-cores-business-in-deal-with-mips/).
- Во-вторых, контроль архитектуры со стороны коммерческих компаний не нравился университетским исследователям. Хотя MIPS активно использовался в учебниках, а ARM давал гранты университетам, но ученые были недовольны перспективой получения писем от корпоративных юристов из-за какого-нибудь созданного ими экспериментального процессора.
- Наконец, во всех RISC-архитектурах скопились разные черты, которые когда-то казались хорошими идеями, но стали тормозом прогресса при усложнении процессоров, повышении частоты, введении микроархитектуры с внеочередным выполнением команд, переменной длины инструкций и предсказателями перехода. У SPARC такой чертой были регистровые окна, у MIPS – слоты отло-

женного ветвления, у ARM – условное выполнение инструкций. Нужна была ревизия мира RISC-процессоров.

И этой ревизией стала RISC-V – архитектура, созданная в 2010 году группой того же Дэвида Паттерсона из Университета Калифорнии в Беркли, который написал два учебника и стоял у истоков архитектуры SPARC. Группа RISC-V не только объединила опыт процессорных компаний за предыдущие 30 лет, но и вступила в партнерство с Linux Foundation и многими крупными компаниями – Google, AMD, Western Digital.

Когда вы используете архитектуру RISC-V для проектирования своего процессора, вам не нужно платить за архитектурную лицензию. При этом сами вы можете получать за свой процессор деньги: продавать его как IP-блок, систему на кристалле или производить на его основе чипы. Вы также можете решить сделать бесплатный процессор с открытым кодом на Verilog для исследователей – это тоже поощряется сообществом вокруг архитектуры RISC-V.

Сейчас RISC-V может сыграть большую роль в становлении российской электроники. Российские компании CloudBEAR и Syntacore (приобретенная компанией «Ядро») работают над процессорами собственной микроархитектуры, совместимыми по системе команд с архитектурой RISC-V. Это идеальная комбинация, которая позволяет разрабатывать свои процессоры и конкурировать по производительности, энергопотреблению и набору расширений с производителями на мировом рынке, одновременно сохраняя программную совместимость со всеми программами, которые создаются для экосистемы RISC-V во всем мире. К таким программам относятся компиляторы, операционные системы и прикладные программы – от программ для миниатюрных чипов для интернета вещей до мобильных устройств, автомобильной электроники, десктопов и суперкомпьютеров.

Подводя итог

Предыдущие издания учебника Харрисов уже помогли исправить серьезный дисбаланс в преподавании цифровой электроники в России, который возник еще в 1990-е годы. Книга также стала отправной точкой для создания курса лабораторных работ на ПЛИС под эгидой МИЭМ НИУ ВШЭ, онлайн-курсов от РОСНАНО и семинаров на ChipEXPO в Сколково. Новое же издание учебника Харрисов выходит как раз тогда, когда в России разворачиваются амбициозные проекты по созданию высокопроизводительных процессорных ядер, которые совместимы с открытой международной архитектурой RISC-V и при этом спроектированы в России.

Мы ожидаем, что читатели этой книги станут топ-разработчиками и бизнес-лидерами российской электронной промышленности и помогут ей занять место в мире, которое соответствует российским традициям достижений в математике, физике, атомных и космических технологиях.

Юрий Панчул,
инженер-проектировщик CPU, GPU и сетевых микросхем,
с опытом работы в MIPS Technologies, Imagination Technologies,
Juniper Networks и Samsung Advanced Computing Lab

Предисловие от редактора русского перевода

Дорогие читатели,
перед вами – уникальное издание.

После распада СССР в русскоязычной образовательной среде возник вакуум, интеллектуальный рынок быстро захватили иностранные САПРы, а на первых ролях оказалась западная электроника. В сфере образования курсы по цифровой электронике нередко сводились к локальным курсам под конкретные платы, наличествовавшие в том или ином университете, а во многих случаях (в том числе из-за отсутствия надлежащего оборудования) учебный процесс превращался в сугубо теоретическое изучение дисциплины. Об этом явлении совершенно справедливо написал Юрий Панчул: https://habr.com/ru/post/589091/ («Почему книга Эндрю Таненбаума "Архитектура компьютера" вредна для образования»), чью точку зрения я полностью поддерживаю, поскольку сам учился по книге Таненбаума.

К счастью, в последующие годы картина начала меняться. Университеты стали богаче, появилась возможность приобретения необходимого оборудования, оно стало доступно и для личного пользования; началось оживление в российских компаниях, и обозначилась все большая потребность в специалистах по цифровой электронике. Все эти обстоятельства сформировали запрос на появление массовых учебных материалов на русском языке.

Звезды сошлись в 2016 году: для написания книги, по инициативе Юрия Панчула, удалось собрать вместе специалистов из ряда университетов и международных компаний, получить финансирование от Imagination technologies и найти понимание ведущего издательства в этой сфере – «ДМК Пресс». Основой для написания материала будущего издания стала великолепная книга D. M. Harris, S. L. Harris «Digital Design and Computer Architecture», де-факто являющаяся стандартом при изучении компьютерной архитектуры и цифрового синтеза во многих зарубежных университетах.

Так появилось первое издание книги «Цифровая схемотехника и архитектура компьютера». Несмотря на то что перевод был в некоторой степени аматорским и в первом издании обнаружились некоторые ошибки и неточности, книга стала бестселлером и разошлась тиражом в не одну тысячу экземпляров. Поскольку перевод первого издания книги осуществлялся без моего участия, мною в учебном процессе использовалась ее английская версия. Но как только появился русский перевод, он был сразу внедрен в учебные курсы, и на нем выросло несколько поколений студентов.

Дальше – больше. «Цифровой синтез» издали в цветном варианте, потом вышло дополнение по архитектуре ARM, а также была выпущена отдельная книга, допол-

няющая основную: «Цифровой синтез: практический курс», которая представляет собой компьютерный практикум, построенный на дешевых и доступных платах ПЛИС, при этом был сделан акцент на изучении языка Verilog.

Следует отметить, что время не стоит на месте: архитектура MIPS, которой посвящена исходная книга, все больше теряет свои позиции и вытесняется RISC-V, объединяющей в себе новые подходы к проектированию RISC-процессоров и принципы открытой разработки. Таким образом, появилась насущная необходимость в переводе нового издания книги D. M. Harris, S. L. Harris «Digital Design and Computer Architecture. RISC-V Edition». Хотя новое издание в целом ряде глав пересекается с исходной книгой, другие главы, посвященные архитектуре RISC-V, – полностью новые. Можно было пойти при этом по одному из путей: либо выпустить дополнение к основной книге (как это было сделано для архитектуры ARM), либо перевыпустить книгу полностью. Чтобы не нарушать целостность произведения, было принято решение пойти по второму пути, попутно исправив допущенные ранее ошибки и тщательно переработав старые главы. Результат этого труда – перед вами.

Данная книга будет полезна всем студентам (таких вузов, как, например, МИЭТ или ИТМО), изучающим архитектуру компьютера и языки описания аппаратуры, а также всем разработчикам, которым необходимо понимать, как устроен микропроцессор / микроконтроллер или другая цифровая схема изнутри.

Александр Юрьевич Романов,
научный редактор книги,
к. т. н., доцент МИЭМ НИУ ВШЭ,
преподаватель курсов «Проектирование систем на кристалле»
и «Системное проектирование цифровых устройств»,
г. Москва, Россия

Предисловие

Эта книга уникальна тем, что описывает цифровую схемотехнику с точки зрения компьютерной архитектуры, начиная с двоичной логики и заканчивая проектированием микропроцессора.

Мы считаем, что проектирование микропроцессора является своеобразным обрядом посвящения для студентов инженерных и компьютерных специальностей. Внутренняя работа микропроцессора кажется почти магической для непосвященных, но при подробном объяснении оказывается простой и доступной для понимания. Проектирование цифровых схем само по себе является захватывающим предметом. Программирование на языке ассемблера позволяет понять внутренний язык, на котором говорит микропроцессор. Микроархитектура, в свою очередь, является тем связующим звеном, которое объединяет эти предметы воедино.

Первые две версии этого набирающего популярность учебника описывают архитектуры MIPS и ARM. MIPS – одна из исходных вычислительных архитектур с сокращенным набором команд (Reduced Instruction Set Computing, RISC), простая в изучении и применении. Значимость архитектуры MIPS сложно переоценить, поскольку она вдохновила разработчиков на создание последующих архитектур, включая RISC-V. Архитектура ARM стала очень популярной за последние несколько десятилетий благодаря своей эффективности и богатой экосистеме. Было продано более 50 млрд процессоров ARM, и более 75 % людей на планете используют продукты с этими процессорами.

В течение последнего десятилетия архитектура RISC-V становится все более значимой как с образовательной, так и с коммерческой точки зрения. Будучи широко распространенной компьютерной архитектурой с открытым исходным кодом, RISC-V сочетает простоту MIPS с гибкостью и функциональностью современных процессоров.

С познавательной точки зрения использование трех версий учебника – MIPS, ARM и RISC-V – полностью идентично. Архитектура RISC-V имеет ряд особенностей, включающих расширяемость и компактный формат представления инструкций, которые повышают ее эффективность, но немного увеличивают сложность. Три микроархитектуры также похожи, а архитектуры MIPS и RISC-V имеют много общего. Мы планируем переиздавать версии учебника про MIPS, ARM и RISC-V до тех пор, пока эти архитектуры востребованы рынком.

Особенности книги

Эта книга содержит ряд особенностей.

Одновременное использование языков SystemVerilog и VHDL

Языки описания аппаратуры (hardware description languages, HDL) находятся в основе современных методов проектирования сложных цифровых систем. К сожалению, разработчики делятся на две примерно равные группы, использующие два

разных языка, – SystemVerilog и VHDL. Языки описания аппаратуры рассматриваются в **главе 4**, сразу после глав, посвященных проектированию комбинационных и последовательностных логических схем. Затем языки HDL используются в **главах 5** и **7** для разработки цифровых блоков большего размера и процессора целиком. Тем не менее **главу 4** можно безболезненно пропустить, если изучение языков HDL не входит в программу.

Эта книга уникальна тем, что использует одновременно и SystemVerilog, и VHDL, что позволяет читателю освоить проектирование цифровых систем сразу на двух языках. В **главе 4** сначала описываются общие принципы, применимые к обоим языкам, а затем вводится синтаксис и приводятся примеры использования этих языков. Этот двуязычный подход облегчает преподавателю выбор языка HDL, а читателю позволит перейти с одного языка на другой как во время учебы, так и в профессиональной деятельности.

Архитектура и микроархитектура процессора RISC-V

Главы 6 и **7** посвящены изучению архитектуры и микроархитектуры RISC-V. Архитектура RISC-V является идеальным учебным пособием в том смысле, что это реальная архитектура, на которой основаны миллионы выпускаемых ежегодно микросхем, и в то же время она проста для изучения. Кроме того, сотни университетов по всему миру разрабатывают учебные курсы, лабораторные работы и различные инструменты именно для этой архитектуры.

Живые примеры

В дополнение к обсуждению основной темы этого учебника – архитектуры RISC-V – в **разделе 6.8** для расширения кругозора студентов рассматривается архитектура процессоров Intel x86. В **главе 9** (доступной в виде онлайн-приложения) также описываются периферийные устройства на примере популярной платы для разработки RED-V RedBoard от SparkFun, в основе которой лежит процессор SiFive Freedom E310 RISC-V. Эти живые примеры показывают, как описанные в данных главах концепции применяются в реальных микросхемах, которые широко используются в персональных компьютерах и бытовой электронике.

Доступное описание высокопроизводительных архитектур

Глава 7 содержит краткий обзор особенностей современных высокопроизводительных микроархитектур, включая такие, как внеочередное выполнение команд, суперскалярность, многопоточность и многоядерность. Материал изложен в доступной для первокурсников форме и показывает, как можно расширить микроархитектуры, описанные в книге, чтобы получить современный процессор.

Упражнения в конце глав и вопросы для собеседования

Лучшим способом изучения цифровой схемотехники является разработка устройств. В конце каждой главы приведены многочисленные упражнения. За

упражнениями следует набор вопросов для собеседования, которые наши коллеги обычно задают студентам, претендующим на работу в отрасли. Эти вопросы предлагают читателю взглянуть на задачи, с которыми соискателям придется столкнуться в ходе собеседования при трудоустройстве. Решения упражнений доступны через веб-сайт книги и специальный веб-сайт для преподавателей. Более подробная информация приведена в следующем разделе.

Материалы в сети Интернет

Дополнительные англоязычные материалы для этой книги доступны на веб-сайте по адресу http://www.ddcabook.com или на сайте издателя: https://www.elsevier.com/books-and-journals/book-companion/9780128200643. Эти веб-сайты доступны для всех читателей и содержат следующие материалы:

- ссылки на видеокурсы;
- решения упражнений с нечетными номерами;
- иллюстрации из книги в форматах PDF и PPTX;
- ссылки на профессиональные инструменты автоматизированного проектирования (САПР) от Intel®;
- инструкции по использованию PlatformIO (расширение Visual Studio Code) для компиляции, сборки и моделирования кода на языках C и ассемблера для процессоров RISC-V;
- HDL-код для процессора RISC-V;
- полезные советы по использованию Intel Quartus;
- слайды лекций в формате PowerPoint (PPTX);
- образцы учебных и лабораторных материалов для курса;
- список опечаток и исправлений.

Также существует специальный веб-сайт для преподавателей, зарегистрировавшихся на https://inspectioncopy.elsevier.com, который содержит:

- решения всех упражнений;
- решения заданий к лабораторным работам.

Открытые курсы на EdX

К этой книге прилагаются открытые курсы на сайте EdX (https://www.edx.org/). Курсы содержат видеолекции, интерактивные упражнения, а также интерактивные наборы задач и лабораторные работы. Набор курсов состоит из двух частей – «Цифровая схемотехника» (ENGR 85A) и «Компьютерная архитектура» (ENGR 85B), – разработанных в Harvey Mudd College (HarveyMuddX; на EdX выполните поиск по фразам «Digital Design HarveyMuddX» и «Computer Architecture HarveyMuddX»). Вам не придется платить за просмотр видео, но EdX взимает плату за интерактивные упражнения и сертификат. Для студентов предусмотрены скидки.

Как использовать программный инструментарий в учебном курсе

Программное обеспечение Intel Quartus

Программное обеспечение Quartus Web Edition и Lite Edition представляет собой бесплатные версии профессиональной САПР Intel Quartus™, предназначенной для разработки устройств на FPGA. Это позволяет студентам проектировать цифровые устройства в виде принципиальных схем или на языках SystemVerilog и VHDL. После создания схемы или кода устройства студенты могут моделировать их поведение с использованием САПР ModelSim™-Intel FPGA Edition или Starter Edition, которые входят в состав САПР Intel Quartus. Quartus также содержит встроенный инструмент логического синтеза, который поддерживает языки описаний SystemVerilog и VHDL.

Разница между Web Edition, Lite Edition и Pro Edition заключается в том, что Web- и Lite Edition поддерживают только подмножество наиболее распространенных FPGA производимых Intel FPGA (Altera). Бесплатные версии ModelSim искусственно снижают производительность моделирования для проектов, содержащих больше 10 тысяч строк HDL-кода, тогда как профессиональная версия ModelSim этого не делает.

PlatformIO

Расширение PlatformIO для редактора Visual Studio Code представляет собой набор средств разработки программного обеспечения (software development kit, SDK) для RISC-V. Поскольку появление каждой новой платформы влекло за собой появление нового SDK, расширение PlatformIO сделало процесс программирования и использования различных процессоров заметно проще благодаря наличию унифицированного интерфейса для большого количества платформ и устройств. SDK PlatformIO можно скачать бесплатно и использовать с RED-V RedBoard SparkFun, как описано в лабораторных работах на веб-сайте[1]. PlatformIO предоставляет доступ к коммерческому компилятору RISC-V и позволяет студентам разрабатывать программы на языках C и ассемблера, компилировать их, а затем запускать и выполнять их отладку на RedBoard SparkFun RED-V (глава 9 и соответствующие лабораторные работы).

Симулятор ассемблера Venus

Симулятор Venus, доступный по адресу https://www.kvakil.me/venus/, – это веб-симулятор ассемблера RISC-V. Программы разрабатываются (или копируются/вставляются) на вкладке **Редактор**, а затем моделируются и запускаются на вкладке **Симулятор**. Во время работы программы можно просматривать содержимое регистров и памяти.

[1] https://docs.platformio.org/en/latest/boards/sifive/sparkfun_redboard_v.html.

Лабораторные работы

Веб-сайт книги содержит ссылки на ряд лабораторных работ, которые охватывают все темы, начиная от проектирования цифровых систем и заканчивая архитектурой компьютера. Из лабораторных работ студенты узнают, как использовать САПР Quartus для описания своих проектов, их моделирования, синтеза и реализации. Лабораторные работы также включают темы по программированию на языке C и языке ассемблера с использованием PlatformIO и RedBoard RED-V от SparkFun.

После синтеза схемы студенты могут реализовать свои проекты, используя платы Altera DE2, DE2-115, DE0 или другую плату FPGA. Лабораторные работы подготовлены для плат DE2 или DE-115. Эти мощные и относительно недорогие платы доступны для заказа на сайте www.de2-115.terasic.com. На платах размещаются микросхемы FPGA, которые можно сконфигурировать для реализации студенческих проектов. Мы предоставляем лабораторные работы, которые описывают, как реализовать различные блоки на плате DE2-115 с помощью программного обеспечения Quartus.

Для проведения лабораторных работ учащимся необходимо загрузить и установить Intel Quartus Web или Lite Edition и Visual Studio Code с расширением PlatformIO. Преподаватели также могут установить эти САПР в учебных лабораториях. Лабораторные работы включают инструкции по разработке проектов на плате DE2/DE2-115. Этап практической реализации проекта на плате можно пропустить, но мы считаем, что он имеет большое значение для получения практических навыков.

Мы протестировали лабораторные работы на ОС Windows, но такие же инструменты доступны и для ОС Linux.

Курсы RVfpga

После изучения материала данной книги мы рекомендуем пройти бесплатный цикл из двух курсов RISC-V FPGA (RVfpga). Первый курс рассказывает о том, как сконфигурировать коммерческое ядро RISC-V для реализации на FPGA, запрограммировать его с помощью языка ассемблера RISC-V или C, добавить к нему периферийные устройства, а также проанализировать и изменить ядро и систему памяти, включая добавление инструкций в ядро. В этом курсе используется система на кристалле (SoC) SweRVolf с открытым исходным кодом (https://github.com/chipsalliance/Cores-SweRVolf), основанная на коммерческом ядре SweRV EH1 от Western Digital (https://www.westerndigital.com/solutions/business/risc-v). В курсе также показано, как использовать симулятор HDL с открытым исходным кодом Verilator и симулятор набора команд RISC-V с открытым исходным кодом Whisper от Western Digital. Второй курс, RVfpga-SoC, показывает, как построить SoC на основе SweRVolf, используя такие функциональные элементы, как ядро SweRV EH1, межмодульные соединения и память. Затем курс рассказывает пользователю о загрузке и запуске операционной системы Zephyr на SoC RISC-V. Все необходимое программное обеспечение и исходный код системы (файлы Verilog/SystemVerilog) бесплатны, а кур-

сы можно проходить с использованием симулятора, поэтому вам не придется покупать оборудование. Материалы RVfpga свободно доступны после регистрации на сайте программы Imagination Technologies University по адресу https://university.imgtec.com/rvfpga/.

Опечатки

Все опытные программисты знают, что любая сложная программа непременно содержит ошибки. Так же происходит и с книгами. Мы старались выявить и исправить все ошибки и опечатки в этой книге. Тем не менее некоторые ошибки могли остаться. Список найденных ошибок будет опубликован на веб-сайте книги.

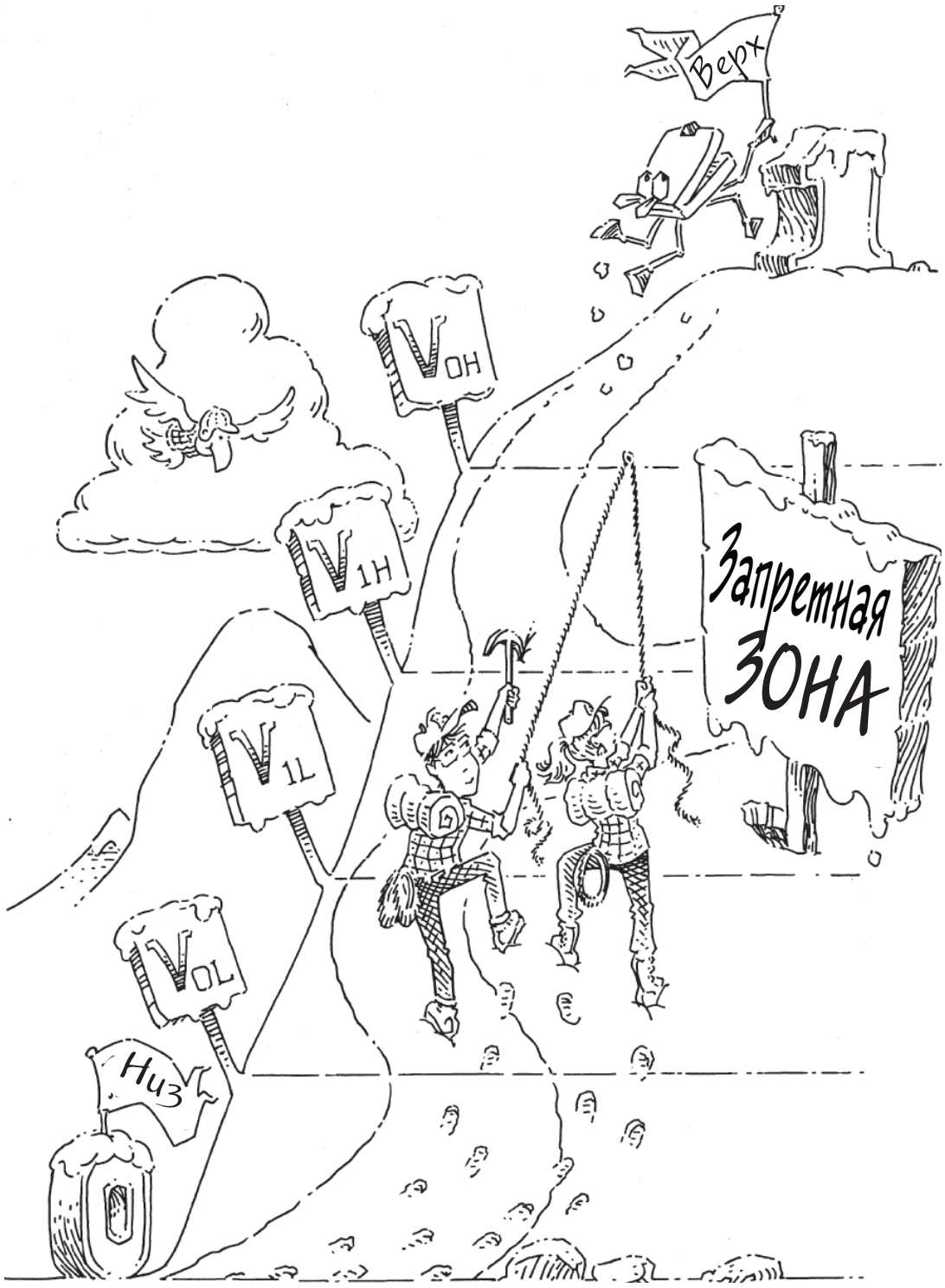
Пожалуйста, присылайте найденные ошибки по адресу ddcabugs@gmail.com (для английской версии книги; для русской версии – присылайте научному редактору русского перевода А. Ю. Романову на электронную почту a.romanov@hse.ru). Первый человек, который сообщит об ошибке в английском издании и предоставит исправление, которое мы используем в будущем переиздании книги, будет вознагражден премией в 1 доллар!

Признательность за поддержку

Мы высоко ценим огромный вклад Стива Меркена (Steve Merken), Нейта Макфаддена (Nate McFadden), Руби Гаммелл (Ruby Gammell), Андрэ Аке (Andrae Akeh), Маникандана Чандрасекарана (Manikandan Chandrasekaran) и остальных членов издательской команды Morgan Kaufmann, которые сделали эту книгу реальностью. Мы любим творчество Дуэйна Бибби (Duane Bibby), чьи забавные рисунки украшают страницы книги.

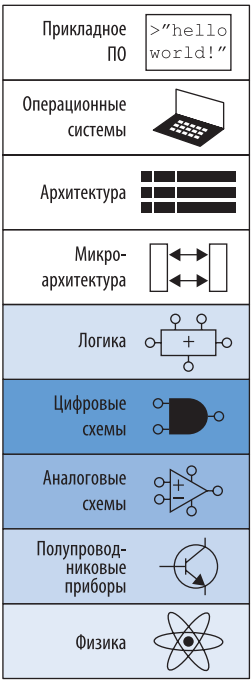
Мы хотели бы поблагодарить Мэтью Уоткинса (Matthew Watkins), который помог написать раздел о гетерогенных многопроцессорных системах в **главе 7**, и Джоша Брейка (Josh Brake), принявшего участие в написании **главы 9** о встроенных системах ввода-вывода. Мы высоко ценим работу Матео Марковича (Mateo Markovic) и Джорди Райдера (Geordie Ryder), которые рецензировали книгу и внесли свой вклад в решения упражнений. Огромный вклад в улучшение качества книги внесли многочисленные рецензенты: Дэниел Чавер Мартинес (Daniel Chaver Martinez), Рой Кравиц (Roy Kravitz), Звонимир Бандич (Zvonimir Bandic), Джузеппе Ди Луна (Giuseppe Di Luna), Штеффен Пол (Steffen Paul), Рави Миттал (Ravi Mittal), Дженнифер Виникус (Jennifer Winikus), Хешам Омран (Hesham Omran), Анхель Солис (Angel Solis), Райнер Дизон (Reiner Dizon) и Олоф Киндгрен (Olof Kindgren). Мы также очень признательны нашим студентам в колледже Harvey Mudd и Университете Невады в Лас-Вегасе, которые дали нам полезные отзывы на черновики этого учебника.

И конечно же, мы оба благодарим наши семьи за их любовь и поддержку.


Верх
V_{OH}
V_{1H}
V_{1L}
V_{OL}
Низ
Запретная
ЗОНА

Глава 1

От нуля до единицы



1.1. План игры

За последние тридцать лет микропроцессоры буквально изменили наш мир до неузнаваемости. Ноутбук сейчас обладает большей вычислительной мощностью, чем большой компьютер из недавнего прошлого, занимавший целую комнату. Внутри современного автомобиля представительского класса можно обнаружить около пятидесяти микропроцессоров. Именно прогресс в области микропроцессорной техники сделал возможным появление сотовых телефонов и сети Интернет, значительно продвинул вперед медицину и радикально изменил тактику и стратегию современных войн. Объем продаж мировой полупроводниковой промышленности вырос с 21 млрд долларов в 1985 году до 300 млрд долларов в 2011 году, причем микропроцессоры составили львиную долю этих продаж. Мы убеждены, что микропроцессоры важны не только с технической, экономической и социальной точек зрения, но и стали одним из самых увлекательных изобретений в истории человечества. Когда вы за-

кончите чтение этой книги, вы будете знать, как разработать и построить ваш собственный микропроцессор, а навыки, полученные на этом пути, пригодятся вам для разработки многих других цифровых систем.

Мы предполагаем, что у вас уже есть базовые знания по теории электричества, некоторый опыт программирования и искреннее желание понять, что происходит внутри компьютера. В этой книге основное внимание уделяется разработке цифровых систем, то есть систем, которые используют для своей работы два уровня напряжения, представляющих единицу и ноль. Мы начнем с простейших цифровых логических элементов (digital logic gates), которые принимают определенную комбинацию единиц и нулей на входах и трансформируют ее в другую комбинацию единиц и нулей на выходах. После этого мы с вами научимся объединять эти простейшие логические элементы в более сложные модули, такие как сумматоры и блоки памяти. Затем перейдем к программированию на языке ассемблера – родном языке микропроцессора. И в завершение из кирпичиков логических элементов мы соберем полноценный микропроцессор, способный выполнять программы, разработанные на языке ассемблера.

Огромным преимуществом цифровых систем над аналоговыми является то, что необходимые для их построения блоки чрезвычайно просты, поскольку оперируют не непрерывными сигналами, а единицами и нулями.

Построение цифровой системы не требует запутанных математических расчетов или глубоких знаний в области физики. Вместо этого задача, стоящая перед разработчиком цифровых устройств, заключается в том, чтобы собрать сложную работающую систему из этих простых блоков. Возможно, микропроцессор станет первой разработанной вами системой, настолько сложной, что ее невозможно целиком удержать в голове. Именно поэтому одной из тем, проходящих красной нитью через эту книгу, является искусство управления сложностью системы.

1.2. Искусство управления сложностью

Одной из характеристик, отличающих профессионального инженера-электронщика или программиста от дилетанта, является систематический подход к управлению сложностью многоуровневой системы. Современные цифровые системы построены из миллионов и миллиардов транзисторов. Человеческий мозг не в состоянии предсказать поведение подобных систем путем составления уравнений, описывающих движение каждого электрона в каждом транзисторе системы, и последующего решения этой системы уравнений. Для того чтобы разработать удачный

микропроцессор и не утонуть при этом в море избыточной информации, необходимо научиться управлять сложностью разрабатываемой системы.

1.2.1. Абстракция

Критически важный принцип управления сложностью системы – *абстракция*, подразумевающая исключение из рассмотрения тех элементов, которые в данном конкретном случае несущественны для понимания работы этой системы. Любую систему можно рассматривать с различных уровней абстракции. Политику, участвующему в выборах, например, нет нужды учитывать все детали окружающего его мира, ему достаточно абстрактной иерархической модели страны, состоящей из населенных пунктов, областей и федеральных округов. В области может быть несколько населенных пунктов, а федеральный округ включает в себя разные области. Если политик борется за пост президента, то его, скорее всего, интересует то, как проголосует федеральный округ в целом, при этом ему не обязательно знать, какое количество голосов он наберет в каждом конкретном населенном пункте этого округа. Для политика федеральный округ – это его уровень абстракции. С другой стороны, бюро переписи населения обязано знать количество жителей в каждом городе или поселке страны и потому должно оперировать на самом низком уровне абстракции данной системы – на уровне населенных пунктов.

На **рис. 1.1** показаны уровни абстракции, типичные для любой электронной компьютерной системы вместе со строительными блоками, характерными для каждого уровня абстракции этой системы. На самом низком уровне абстракции находится физика, изучающая движение электронов. Поведение электронов описывается квантовой механикой и системой уравнений Максвелла.

Рассматриваемая нами современная электронная система состоит из полупроводниковых устройств (devices), таких как транзисторы (а когда-то это были электронные лампы). Каждое такое устройство имеет четко определенные точки соединения с другими подобными устройствами. Эти точки мы будем называть *контактами* (в англоязычной литературе используется термин *terminal*). Любое электронное устройство может быть представлено абстрактной математической моделью, описывающей изменяющуюся во времени взаимозависимость тока и напряжения. Такие же изменения тока и напряжения можно наблюдать на экране осциллографа, если подключить осциллограф к контактам реального устройства. Данный под-

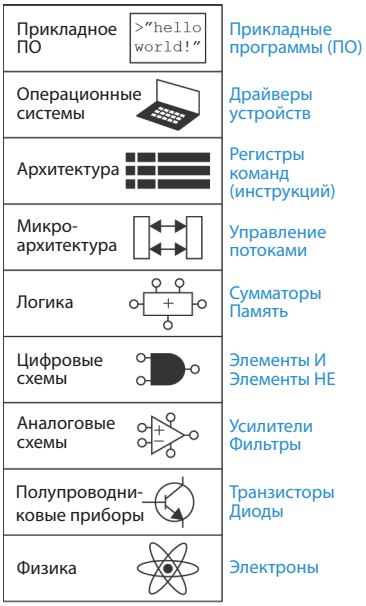


Рис. 1.1 Уровни абстракции электронной вычислительной системы

Каждая глава этой книги начинается с иконок (рис. 1.1), символически изображающих уровни абстракции электронной системы, которые мы перечислили выше. Иконка темно-синего цвета указывает на тот уровень абстракции, которому уделяется главное внимание в этой конкретной главе. Иконки более светлого оттенка синего указывают на другие уровни абстракции, также затронутые в главе.

ход означает, что если рассматривать систему на уровне устройств, функции которых однозначно определены, то можно не учитывать поведение электронов внутри отдельных устройств этой системы.

Следующий уровень абстракции – это *аналоговые схемы* (analog circuits), в которых полупроводниковые устройства соединены таким образом, чтобы они образовывали функциональные компоненты, например усилители. Напряжение на входе и на выходе аналоговой цепи изменяется в непрерывном диапазоне.

В отличие от аналоговых цепей, *цифровые схемы* (digital circuits), такие как логические элементы, используют два строго ограниченных дискретных уровня напряжения. Один из этих дискретных уровней – это логический ноль, другой – логическая единица. В разделах этой книги, посвященных разработке цифровых схем и устройств, мы будем использовать простейшие цифровые схемы для построения сложных цифровых модулей, таких как сумматоры и блоки памяти.

Микроархитектурный уровень абстракции, или просто *микроархитектура* (microarchitecture), связывает логический и архитектурный уровни абстракции. Архитектурный уровень абстракции, или *архитектура* (architecture), описывает компьютер с точки зрения программиста. Например, архитектура Intel x86, используемая микропроцессорами большинства персональных компьютеров (ПК), определяется набором инструкций и регистров (памяти для временного хранения переменных), доступным для использования программистом. Микроархитектура – это соединение простейших цифровых элементов в логические блоки, предназначенные для выполнения команд, определенных какой-то конкретной архитектурой. Отдельно взятая архитектура может быть реализована с использованием различных вариантов микроархитектур с разным соотношением цены, производительности и потребляемой энергии, и такое соотношение зачастую выбирается как баланс между этими тремя факторами. Процессоры Intel Core i7, Intel 80486 и AMD Athlon, например, используют одну и ту же архитектуру x86, но реализованную с применением трех разных микроархитектурных решений.

Рассмотрим область программного обеспечения. *Операционная система* (operating system) управляет операциями нижнего уровня, такими как доступ к жесткому диску или управление памятью. И наконец, программное обеспечение использует ресурсы операционной системы для решения конкретных задач пользователя.

Именно принцип *абстрагирования от маловажных деталей* позволяет вашей бабушке общаться с внуками в интернете, не задумываясь о квантовых колебаниях электронов или организации памяти компьютера.

Предмет этой книги – уровни абстракции от цифровых схем до компьютерной архитектуры. Работая на каком-либо из этих уровней абстракции, полезно знать кое-что и об уровнях абстракции, непосредственно сопряженных с тем уровнем, где вы находитесь. Программист, например, не сможет полностью оптимизировать код без понимания архитектуры процессора, который будет выполнять эту программу. Инженер-электронщик, разрабатывающий какой-либо блок микросхемы, не сможет найти компромисс между быстродействием и уровнем потребления энергии транзисторами, ничего не зная о той цифровой схеме, где этот блок будет использоваться. Мы надеемся, что к тому времени, когда вы закончите чтение этой книги, вы сможете выбрать уровень абстракции, необходимый для успешного выполнения любой стоящей перед вами задачи, и оценить влияние ваших инженерных решений на другие уровни абстракции в разрабатываемой вами системе.

1.2.2. Конструкторская дисциплина

Конструкторская дисциплина – это преднамеренное ограничение самим конструктором выбора возможных вариантов разработки, что позволяет работать продуктивнее на более высоком уровне абстракции. Использование взаимозаменяемых частей – это, вероятно, самый хорошо знакомый всем нам пример практического применения конструкторской дисциплины. Одним из первых примеров использования взаимозаменяемых деталей и узлов стала унификация при производстве кремневых ружей. До начала XIX века такие ружья производились вручную и в штучном порядке. Высококвалифицированный оружейный мастер тщательно подтачивал и подгонял комплектующие, произведенные несколькими не связанными друг с другом ремесленниками. Конструкторская дисциплина для обеспечения взаимозаменяемости деталей и узлов произвела революцию в оружейной промышленности. Ограничение ассортимента комплектующих деталей до стандартного набора с жестко установленными допусками для каждой детали позволило собирать и ремонтировать ружья гораздо быстрее и использовать при этом менее квалифицированный персонал. Оружейный мастер перестал тратить свое время на разрешение проблем, связанных с нижними уровнями абстракции, такими как доводка какого-то конкретного ствола или исправление формы отдельного взятого приклада.

В контексте данной книги соблюдение конструкторской дисциплины в виде максимального использования цифровых схем играет очень важную роль. В цифровых схемах используются дискретные значения напряжения, в то время как в аналоговых схемах напряжение изменяется непрерывно. Таким образом, цифровые схемы, которые можно рассматривать как подмножество аналоговых цепей, в некотором смысле уступают по своим характеристикам более широкому классу аналого-

вых цепей. Но цифровые цепи гораздо проще разработать. Ограничивая использование аналоговых схем и по возможности заменяя их цифровыми, мы можем легко объединять отдельные компоненты в сложные системы, которые в конечном итоге для большинства приложений превзойдут по своим параметрам системы, построенные на аналоговых цепях. Примером тому могут служить цифровые телевизоры, компакт-диски (CD) и мобильные телефоны, которые уже практически полностью вытеснили своих аналоговых предшественников.

Капитан Мериуэзер Льюис — один из руководителей знаменитой экспедиции Льюиса и Кларка на северо-запад США, был, пожалуй, одним из самых ранних сторонников взаимозаменяемости. В 1806 году в своем дневнике, описывая особенности кремниевых унификации деталей кремневых ружей того времени, он написал следующее:

> «Ружья Дрюера и сержанта Прайора одновременно вышли из строя. На ружье Дрюера сломался ударно-спусковой механизм, и мы заменили его на новый. У ружья сержанта Прайора был сломан курковый винт, вместо которого мы поставили запасной, заранее изготовленный специально для ударно-спускового механизма этого ружья на мануфактуре Харперс Фейри, где это оружие и было произведено. Если бы не предусмотрительность, заключавшаяся в том, что мы заранее позаботились о запасных частях для ружей, и не мастерство Джона Шилдса, выполнившего всю работу, то большинство ружей нашей экспедиции к этому времени было бы полностью непригодно для какого-либо использования. И я имею полное право записать в своем дневнике, что, к счастью для нас, все наше оружие находится в прекрасном состоянии».

См.: История экспедиции Льюиса и Кларка: в 4 т. / под ред. Элиота Куэса. 1-е изд.: Харпер, Нью-Йорк, 1893; Переизд.: Довер, Нью-Йорк (3 тома), 3:817.

1.2.3. Три базовых принципа

В дополнение к абстрагированию от несущественных деталей и конструкторской дисциплине разработчики электронных систем используют еще три базовых принципа для управления сложностью системы: иерархичность, модульность конструкции и регулярность. Эти принципы применимы как к программному обеспечению, так и к аппаратной части компьютерных систем.

- *Иерархичность* – принцип иерархичности предполагает разделение системы на отдельные модули, а затем последующее разделение каждого такого модуля на фрагменты до уровня, позволяющего легко понять поведение каждого конкретного фрагмента.
- *Модульность* – принцип модульности требует, чтобы каждый модуль в системе имел четко определенную функциональность и набор интерфейсов и мог быть легко и без непредвиденных побочных эффектов соединен с другими модулями системы.
- *Регулярность* – принцип регулярности требует соблюдения единообразия при разработке отдельных модулей системы. Стандартные модули общего назначения, например такие как блоки питания, могут использоваться многократно, во много раз снижая количество модулей, необходимых для разработки новой системы.

Для иллюстрации трех базовых принципов вновь воспользуемся аналогией из оружейного производства. Нарезное кремневое ружье было одним из самых сложных устройств массового применения в начале XIX века. Используя принцип иерархичности, мы можем разделить его на три главных модуля, как показано на **рис. 1.2**: ствол, ударно-спусковой механизм и приклад с цевьем.

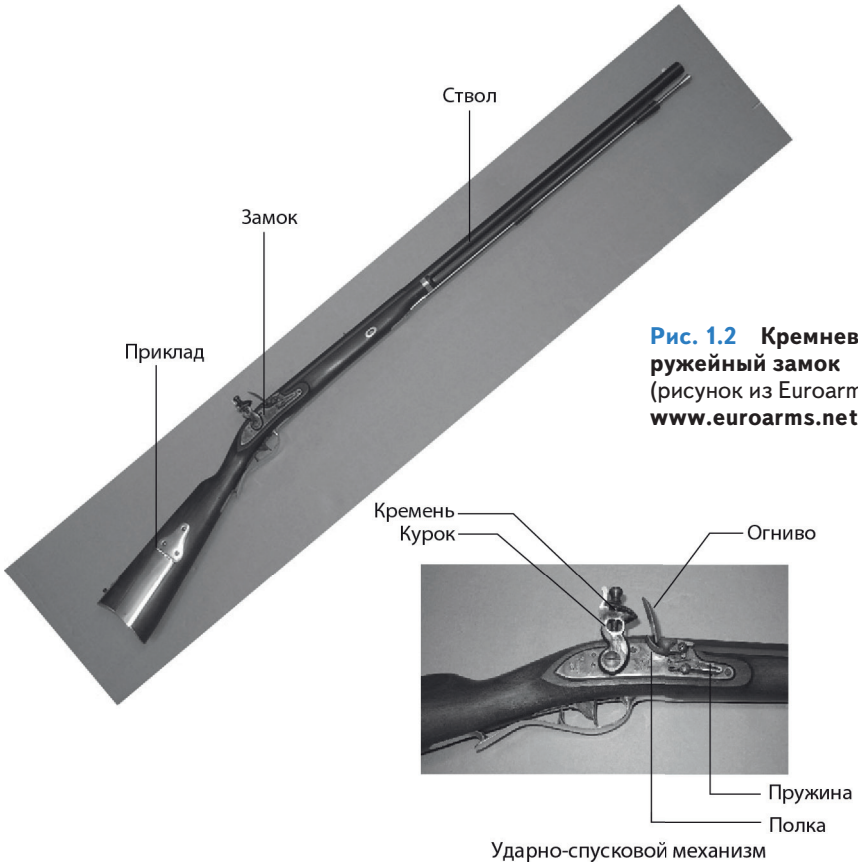


Рис. 1.2 Кремневый ружейный замок (рисунок из Euroarms Italia **www.euroarms.net** © 2006 г.)

Ствол – это длинная металлическая труба, через которую при выстреле выбрасывается пуля. Ударно-спусковой механизм производит выстрел. Деревянные приклад и цевье соединяют воедино остальные части ружья и обеспечивают стрелку надежное удержание оружия при выстреле. В свою очередь, ударно-спусковой механизм включает в себя спусковой крючок, курок, кремень, огниво и пороховую полку. Каждый из этих компонентов также может рассматриваться как следующий иерархический уровень и может быть разделен на более мелкие детали.

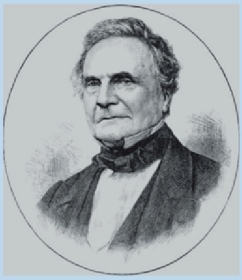
Принцип модульности требует, чтобы каждый компонент выполнял четко определенную функцию и имел интерфейс. Функция приклада и цевья – служить базой для установки ствола и ударно-спускового механизма. Интерфейс для приклада и цевья – это их длина и расположение крепежных элементов, таких как винты или шурупы. Ствол ружья, изготовленного с соблюдением принципа модульности конструкции, может быть установлен на приклады и цевья от разных производителей, если все соединяемые части имеют правильную длину и подходящие

крепежные элементы. Функция ствола – разогнать пулю до необходимой скорости и придать ей вращение, чтобы увеличить точность стрельбы[1]. Принцип модульности требует также, чтобы при соединении модулей не возникало никаких побочных эффектов: конструкция приклада и цевья не должна препятствовать функционированию ствола.

Принцип регулярности учит тому, что взаимозаменяемые детали – это хорошая идея. При соблюдении принципа регулярности поврежденный ствол может быть с легкостью заменен на аналогичный. Стволы могут изготовляться на поточной линии с гораздо большей экономической эффективностью, чем в случае штучного производства.

В данной книге мы будем постоянно возвращаться к этим трем базовым принципам: иерархичности, модульности и регулярности.

1.3. Цифровая абстракция



Чарльз Бэббидж
1791–1871

Чарльз Бэббидж родился в 1791 году. Закончил Кембриджский университет и женился на Джорджиане Витмур. Он изобрел аналитическую машину – первый в мире механический компьютер. Чарльз Бэббидж также изобрел предохранительную решетку для локомотивов, спидометр и универсальный почтовый тариф. Ученый также очень интересовался отмычками для замков и почему-то ненавидел уличных музыкантов. (Портрет любезно предоставлен Fourmilab Швейцария, www.fourmilab.ch.)

Большинство физических величин изменяются непрерывно. Например, напряжение в электрическом проводе, частота колебаний или распределение массы – все это параметры, изменяющиеся непрерывно. Цифровые системы, с другой стороны, представляют информацию в виде дискретно меняющихся переменных с конечным числом строго определенных значений.

Одной из наиболее ранних цифровых систем стала аналитическая машина Чарльза Бэббиджа, которая использовала переменные с десятью дискретными значениями. Начиная с 1834 года и до 1871 года[2] Бэббидж разрабатывал и пытался построить этот механический компьютер. Шестеренки аналитической машины могли находиться в одном из десяти фиксированных положений, а каждое такое положение было промаркировано от 0 до 9, подобно механическому счетчику пробега автомобиля. **Рисунок 1.3** показывает, как выглядел прототип аналитической машины. Каждый ряд шестеренок такой машины обрабатывал одну цифру. В своем механическом компьютере Бэббидж использовал 25 рядов шестеренок таким образом, чтобы машина обеспечивала вычисления с точностью до 25-го знака.

В отличие от машины Бэббиджа большинство электронных компьютеров использует двоичный (бинарный) код. В случае двоичного кода высокое напряжение – это единица, а низкое напряжение – ноль, поскольку гораздо легче оперировать двумя уровнями напряжения, чем десятью.

---

[1] Кремневые ружья не были нарезными и использовали круглые пули. – *Прим. перев.*

[2] А большинству из нас кажется, что обучение в университете – это так долго!

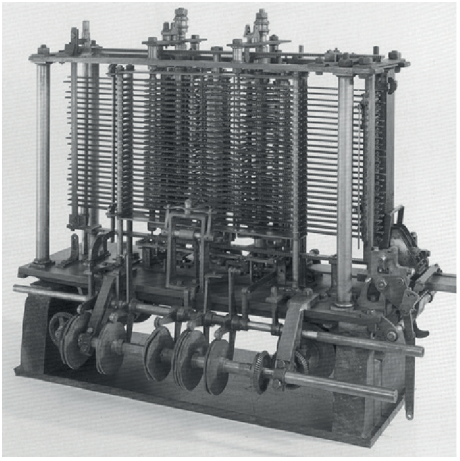


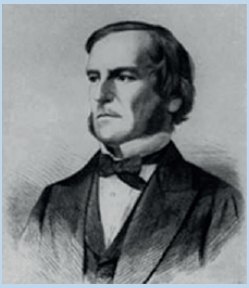
Рис. 1.3 Аналитическая машина Бэббиджа в год его смерти (1871)
(изображение любезно предоставлено Музеем науки и общества)

Объем информации D, передаваемый одной дискретной переменной, которая может находиться в N различных состояниях, измеряется в единицах, называемых *битами,* и вычисляется по следующей формуле:

$$D = \log_2 N \text{ бит.} \tag{1.1}$$

Двоичная переменная передает $\log_2 2 = 1$ – один бит информации. Теперь вам, вероятно, понятно, почему единица информации называется битом. Бит (bit) – это сокращение от английского *binary digit*, что дословно переводится как *двоичный разряд*. Каждая шестеренка в машине Бэббиджа содержит $\log_2 10 = 3{,}322$ бита информации, поскольку она может находиться в одном из $2^{3{,}322} = 10$ уникальных положений. Теоретически непрерывный сигнал может передавать бесконечное количество информации, поскольку может принимать неограниченное число значений. На практике шум и ошибки измерения ограничивают информацию, передаваемую большинством непрерывных сигналов, диапазоном от 10 бит до 16 бит. Если же измерение уровня сигнала должно быть произведено очень быстро, то объем передаваемой информации будет еще ниже (в случае 10 бит, например, это будет только 8 бит).

Предмет этой книги – цифровые схемы, использующие двоичные переменные ноль и единицу. Джордж Буль разработал систему логики, использующую двоичные переменные, и эту систему сегодня называют его именем – *булева логика*. Логические переменные могут принимать значения *ИСТИНА* (TRUE) или *ЛОЖЬ* (FALSE). В электронных компьютерах положительное напряжение обычно представляет единицу, а нулевое напряжение представляет ноль. В этой книге мы будем использовать понятия единица (1), ИСТИНА (TRUE) и ВЫСОКИЙ УРОВЕНЬ СИГНАЛА (HIGH) как синонимы. Аналогичным образом мы будем использовать ноль (0), ЛОЖЬ (FALSE) и НИЗКИЙ УРОВЕНЬ СИГНАЛА (LOW) как взаимозаменяемые термины.



Джордж Буль
1815–1864

Джордж Буль родился в семье небогатого ремесленника. Родители Джорджа не могли оплатить его формального образования, поэтому он осваивал математику самоучкой. Несмотря на это, Булю удалось стать преподавателем Королевского колледжа в Ирландии. В 1854 году Джордж Буль написал свою работу «Исследование законов мышления», которая впервые ввела в научный оборот двоичные переменные, а также три основных логических оператора И, ИЛИ, НЕ (AND, OR, NOT). (Портрет любезно предоставлен Американским физическим институтом.)

Преимущества *цифровой абстракции* заключаются в том, что разработчик цифровой системы может сосредоточиться исключительно на единицах и нулях, полностью игнорируя, каким образом логические переменные представлены на физическом уровне. Разработчика не волнует, представлены ли нули и единицы определенными значениями напряжения, вращающимися шестернями или уровнем гидравлической жидкости. Программист может продуктивно работать, не располагая детальной информацией об аппаратном обеспечении компьютера. Но понимание того, как работает это аппаратное обеспечение, позволяет программисту гораздо лучше оптимизировать программу для конкретного компьютера.

Как вы могли видеть выше, один-единственный бит не может передать большого количества информации. Поэтому в следующем разделе мы рассмотрим вопрос о том, каким образом набор битов можно использовать для представления десятичных чисел. В последующих главах мы также покажем, как группы битов могут представлять буквы и даже целую программу.

1.4. Системы счисления

Все мы привыкли работать с десятичными числами. Но в цифровых системах, построенных на единицах и нулях, использование двоичных или шестнадцатеричных чисел зачастую более удобно. В данном разделе мы рассмотрим системы счисления, использованные в этой книге.

1.4.1. Десятичная система счисления

Еще в начальной школе всех нас научили считать и выполнять различные арифметические операции в *десятичной* (decimal) системе счисления. Такая система использует десять арабских цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 – столько же, сколько у нас пальцев на руках. Числа больше 9 записываются в виде строки цифр. Причем цифра, находящаяся в каждой последующей позиции такой строки, начиная с крайней правой цифры, имеет «вес», в десять раз превышающий «вес» цифры, находящейся в предыдущей позиции. Именно поэтому десятичную систему счисления называют *системой по основанию* (base) 10. Справа налево «вес» каждой позиции увеличивается следующим образом: 1, 10, 100, 1000 и т. д. Позицию, которую цифра занимает в строке десятичного числа, называют разрядом, или декадой.

Чтобы избежать недоразумений при одновременной работе с более чем одной системой счисления, основание системы обычно указывается путем добавления цифры позади и чуть ниже основного числа: 9742_{10}. **Рисунок 1.4** показывает, для примера, как десятичное число 9742_{10} может быть записано в виде суммы цифр, составляющих это число, умноженных на «вес» разряда, соответствующего каждой конкретной цифре.

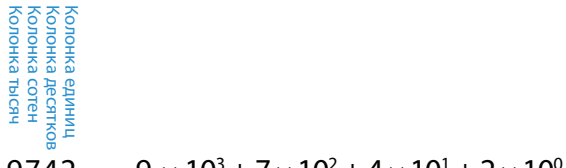


Рис. 1.4 Представление десятичного числа

N-разрядное десятичное число может представлять одну из 10^N цифровых комбинаций: 0, 1, 2, 3, ... $10^N - 1$. Это называется диапазоном *N*-разрядного числа. Десятичное число, состоящее из трех цифр (разрядов), например, представляет одну из 1000 возможных цифровых комбинаций в диапазоне от 0 до 999.

1.4.2. Двоичная система счисления

Одиночный бит может принимать одно из двух значений, 0 или 1. Несколько битов, соединенных в одной строке, образуют *двоичное* (binary) число. Каждая последующая позиция в двоичной строке имеет вдвое больший «вес», чем предыдущая позиция, так что двоичная система счисления – это система по основанию 2. В двоичном числе «вес» каждой позиции увеличивается (так же, как и в десятичном – справа налево) следующим образом: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16 384, 32 768, 65 536 и т. д. Работая с двоичными числами, очень полезно для экономии времени запомнить значения степеней двойки до 2^{16}.

Произвольное *N*-разрядное двоичное число может представлять одну из 2^N цифровых комбинаций: 0, 1, 2, 3, ... $2^N - 1$. В **табл. 1.1** собраны 1-битные, 2-битные, 3-битные и 4-битные двоичные числа и их десятичные эквиваленты.

Пример 1.1 ПРЕОБРАЗОВАНИЕ ЧИСЕЛ ИЗ ДВОИЧНОЙ СИСТЕМЫ СЧИСЛЕНИЯ В ДЕСЯТИЧНУЮ

Преобразовать двоичное число 10110_2 в десятичное.

Решение Необходимые преобразования представлены на **рис. 1.5**.

$$10110_2 = 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0 = 22_{10}$$

Одна шестнадцать · Нет восемь · Одна четыре · Одна двойка · Нет единиц

Рис. 1.5 Преобразование двоичного числа в десятичное число

Таблица 1.1 Таблица двоичных чисел и их десятичный эквивалент

1-битные двоичные числа	2-битные двоичные числа	3-битные двоичные числа	4-битные двоичные числа	Десятичные эквиваленты
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Пример 1.2 ПРЕОБРАЗОВАНИЕ ЧИСЕЛ ИЗ ДЕСЯТИЧНОЙ СИСТЕМЫ СЧИСЛЕНИЯ В ДВОИЧНУЮ

Преобразовать десятичное число 84_{10} в двоичное.

Решение Определите, что должно стоять в каждой позиции двоичного результата: 1 или 0. Вы можете делать это, начиная с левой или правой позиции.

Если начать слева, найдите наибольшую степень 2, меньшую или равную заданному числу (в примере такая степень – это 64). 84 > 64, поэтому ставим 1 в позиции, соответствующей 64. Остается 84 – 64 = 20, 20 < 32, так что в позиции 32 надо поставить 0, 20 > 16, поэтому в позиции 16 ставим 1. Остается 20 – 16 = 4. 4 < 8, поэтому 0 в позиции 8. 4 ≥ 4 – ставим 1 в позицию 4. 4 – 4 = 0, поэтому будут 0 в позициях 2 и 1. Собрав все вместе, получаем $84_{10} = 1010100_2$.

Если начать справа, будем последовательно делить исходное число на 2. Остаток идет в очередную позицию. 84/2 = 42, поэтому 0 в самой правой позиции. 42/2 = 21, 0 во вторую позицию. 21/2 = 10, остаток 1 идет в позицию, соот-

ветствующую 4. $10/2 = 5$, поэтому 0 в позицию, соответствующую 8. $5/2 = 2$, остаток 1 в позицию 16. $2/2 = 1$, 0 в 32 позицию. Наконец, $1/2 = 0$ с остатком 1, который идет в позицию 64. Снова $84_{10} = 1010100_2$.

1.4.3. Шестнадцатеричная система счисления

Использование длинных двоичных чисел для записи и выполнения математических расчетов на бумаге утомительно и чревато ошибками. При этом длинное двоичное число можно разбить на группы по четыре бита, каждая из которых представляет одну из $2^4 = 16$ цифровых комбинаций. Именно поэтому зачастую бывает удобнее использовать для работы систему счисления по основанию 16, называемую *шестнадцатеричной* (hexadecimal). Для записи шестнадцатеричных чисел используются цифры от 0 до 9 и буквы от A до F, как показано в **табл. 1.2**. В шестнадцатеричном числе «вес» каждой позиции меняется следующим образом: 1, 16, 16^2 (или 256), 16^3 (или 4096) и т. д.

Интересно, что термин *hexadecimal* (шестнадцатеричный) введен в научный обиход корпорацией IBM в 1963 году и является комбинацией греческого слова *hexi* (шесть) и латинского decem (десять). Правильнее было бы использовать латинское же слово *sexa* (шесть), но термин *sexadecimal* воспринимался бы несколько неоднозначно.

Таблица 1.2 Шестнадцатеричная система счисления

Шестнадцатеричная цифра	Десятичный эквивалент	Двоичный эквивалент
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Пример 1.3 ПРЕОБРАЗОВАНИЕ ШЕСТНАДЦАТЕРИЧНОГО ЧИСЛА В ДВОИЧНОЕ И ДЕСЯТИЧНОЕ

Преобразовать шестнадцатеричное число $2ED_{16}$ в двоичное и десятичное.

Решение Преобразование шестнадцатеричного числа в двоичное и обратно – очень простое, так как каждая шестнадцатеричная цифра прямо соответствует

4-разрядному двоичному числу. $2_{16} = 0010_2$, $E_{16} = 1110_2$ и $D_{16} = 1101_2$, так что $2ED_{16} = 001011101101_2$. Преобразование в десятичную систему счисления требует выполнения арифметических операций, показанных, показанной на **рис. 1.6**.

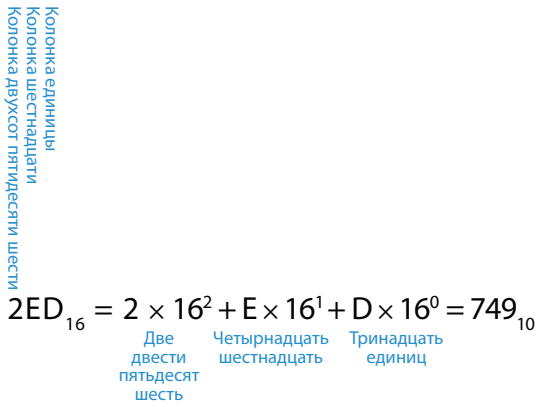


Рис. 1.6 Преобразование шестнадцатеричного числа в десятичное число

Пример 1.4 ПРЕОБРАЗОВАНИЕ ДВОИЧНОГО ЧИСЛА В ШЕСТНАДЦАТЕРИЧНОЕ

Преобразовать двоичное число 1111010_2 в шестнадцатеричное.

Решение Повторим еще раз, это просто. Начинаем справа. 4 наименее значимых бита $1010_2 = A_{16}$. Следующие биты $111_2 = 7_{16}$. Отсюда $1111010_2 = 7A_{16}$.

Пример 1.5 ПРЕОБРАЗОВАНИЕ ДЕСЯТИЧНОГО ЧИСЛА В ШЕСТАНДЦАТЕРИЧНОЕ И ДВОИЧНОЕ

Преобразовать десятичное число 333_{10} в шестнадцатеричное и двоичное.

Решение Как и в случае преобразования десятичного числа в двоичное, можно начать как слева, так и справа.

Если начать слева, найдите наибольшую степень шестнадцати, меньшую или равную заданному числу (в нашем случае это $16^2 = 256$). Число 256 содержится в числе 333 только один раз, поэтому в позицию с «весом» 256 мы записываем единицу. Остается число $333 - 256 = 77$. Число 16 содержится в числе 77 четыре раза, поэтому в позицию с «весом» 16 записываем четверку. Остается $77 - 16 \times 4 = 13$. $13_{10} = D_{16}$, поэтому в позицию с «весом» 1 записываем цифру D. Итак, $333_{10} = 14D_{16}$, это число легко преобразовать в двоичное, как мы показали в примере 1.3: $14D_{16} = 101001101_2$.

Если начинать справа, будем повторять деление на 16. Каждый раз остаток идет в очередную колонку. $333/16 = 20$ с остатком $13_{10} = D_{16}$, который идет в самую правую позицию. $20/16 = 1$ с остатком 4, который идет в позицию с «весом» 16. $1/16 = 0$ с остатком 1, который идет в позицию с «весом» 256. В результате опять получаем $14D_{16}$.

1.4.4. Байт, полубайт и «весь этот джаз»

Группа из восьми бит называется *байт* (byte). Байт представляет $2^8 = 256$ цифровых комбинаций. Размер модулей, сохраненных в памяти компьютера, обычно измеряется именно в байтах, а не битах.

Группа из четырех бит (половина байта) называется *полубайт* (nibble). Полубайт представляет $2^4 = 16$ цифровых комбинаций. Одна шестнадцатеричная цифра занимает один полубайт, а две шестнадцатеричные цифры – один байт. В настоящее время полубайты уже не находят широкого применения, но этот термин все же стоит знать, да и звучит он забавно (в англ. языке *nibble* означает откусывать что-либо маленькими кусочками).

Микропроцессор обрабатывает данные не целиком, а небольшими блоками, называемыми словами. Размер *слова* (word) не является величиной, установленной раз и навсегда, а определяется архитектурой каждого конкретного микропроцессора. На момент написания этой главы (в 2012 году) абсолютное большинство компьютеров использовало 64-битные процессоры. Такие процессоры обрабатывают информацию блоками (словами) длиной 64 бита. А еще не так давно верхом совершенства считались компьютеры, обрабатывающие информацию словами длиной 32 бита. Интересно, что и сегодня наиболее простые микропроцессоры и особенно те, что управляют работой таких бытовых устройств, как, например, тостеры или микроволновые печи, используют слова длиной 16 бит или даже 8 бит.

Если подходить абсолютно строго к терминологии, то микропроцессором называется такой процессор, все элементы которого размещаются на одной микросхеме. До 70-х годов XX века полупроводниковая технология не позволяла разместить процессор целиком на одной микросхеме, поэтому процессоры мощных компьютеров представляли собой набор плат с довольно большим количеством различных микросхем на них. Компания Intel в 1971 году представила первый 4-битный микропроцессор, получивший в качестве названия номер 4004. В наши дни даже самые передовые суперкомпьютеры построены на микропроцессорах, поэтому в этой книге мы будем считать «микропроцессор» и «процессор» тождественными понятиями и использовать оба этих термина как синонимы.

В рамках одной группы битов конечный бит, находящийся на одном конце этой группы (обычно правом), называется *наименее значимым битом* (least significant bit, LSB), или просто младшим битом, а бит на другом конце группы называется *наиболее значимым битом* (most significant bit, MSB), или *старшим битом*. **Рисунок 1.7 (a)** демонстрирует наименее и наиболее значимые биты в случае 6-битного двоичного числа. Аналогичным образом внутри одного слова можно выделить *наименее значимый байт* (least significant byte, LSB), или младший байт, и *наиболее значимый байт* (most significant byte, MSB), или старший байт. **Рисунок 1.7 (b)** показывает, как это делается в случае 4-байтного числа, записанного восемью шестнадцатеричными цифрами.

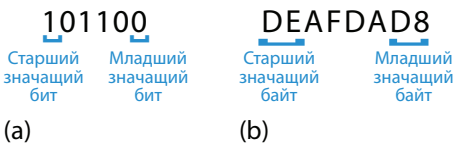


Рис. 1.7 Наименее и наиболее значимые биты и байты

В силу удачного совпадения $2^{10} = 1024 \approx 10^3$. Этот факт позволяет нам использовать приставку *кило* (греческое название тысячи) для сокращенного обозначения 2^{10}. Например, 2^{10} байт – это один килобайт (1 КБ). Подобным же образом *мега* (греческое название миллиона) обозначает $2^{20} \approx 10^6$, а *гига* (греческое название миллиарда) указывает на $2^{30} \approx 10^9$. Зная, что $2^{10} \approx 1$ тысяча, $2^{20} \approx 1$ миллион, $2^{30} \approx 1$ миллиард и помня значения степеней двойки до 2^9 включительно, будет легко приблизительно рассчитать в уме любую другую степень двух.

Пример 1.6 ОЦЕНКА СТЕПЕНЕЙ ДВОЙКИ

Найдите приблизительное значение 2^{24} без использования калькулятора.

Решение Представьте экспоненту как число, кратное десяти, и остаток.

$2^{24} = 2^{20} \times 2^4$, $2^{20} \approx 1$ миллион. $2^4 = 16$. Итак, $2^{24} \approx 16$ миллионов. На самом деле $2^{24} = 16\,777\,216$, но 16 миллионов – достаточно хорошее приближение для маркетинговых целей.

Так же как 1024 байта называют *килобайтом* (КБ), 1024 бита называют *килобитом* (Кб или кбит). Аналогичным образом МБ, Мб, ГБ и Гб используются для сокращенного обозначения миллиона и миллиарда байт и бит. Размеры элементов памяти обычно измеряются в байтах. А вот скорость передачи данных измеряется в битах в секунду. Максимальная скорость передачи данных телефонным модемом, например, составляет 56 килобит в секунду.

1.4.5. Сложение двоичных чисел

Сложение двоичных чисел производится так же, как и сложение десятичных, с той лишь разницей, что двоичное сложение выполнить гораздо проще (**рис. 1.8**). Как и при сложении десятичных чисел, если сумма двух чисел превышает значение, помещающееся в один разряд, мы переносим 1 в следующий разряд. На **рис. 1.8** для сравнения показано сложение десятичных и двоичных чисел. В крайней правой колонке на **рис. 1.8 (a)** складываются числа 7 и 9. Сумма 7 + 9 = 16, что превышает 9, а значит, больше того, что может вместить один десятичный разряд. Поэтому мы записываем в первый разряд 6 (первая колонка) и переносим 10 в следующий разряд (вторая колонка) как 1. Аналогичным же образом при сложении двоичных чисел, если сумма двух чисел превышает 1, мы переносим 2 в следующий разряд как 1. В правой колонке на **рис. 1.8 (b)**, например, сумма $1 + 1 = 2_{10} = 10_2$, что не может уместиться в одном двоичном разряде. Поэтому мы записываем 0 в первом разряде (первая колонка) и 1 в следующем разряде (вторая колонка). Во второй колонке опять складываются 1 и 1 и еще добавляется 1, перенесенная сюда после сложения чисел в первой колонке. Сумма $1 + 1 + 1 = 3_{10} = 11_2$.

Мы записываем 1 в первый разряд (вторая колонка) и снова добавляем 1 в следующий разряд (третья колонка). По очевидной причине бит, добавленный в соседний разряд (колонку), называется *битом переноса* (carry bit).

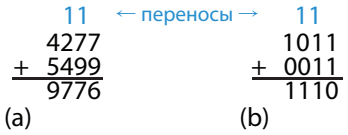


Рис. 1.8 Примеры сложения с переносом: (a) десятичное, (b) двоичное

Пример 1.7 ДВОИЧНОЕ СЛОЖЕНИЕ

Вычислить $0111_2 + 0101_2$.

Решение На **рис. 1.9** показано, что сумма равна 1100_2. Переносы выделены синим цветом. Мы можем проверить нашу работу, повторив вычисления в десятичной системе счисления. $0111_2 = 7_{10}$, $0101_2 = 5_{10}$. Сумма равна $12_{10} = 1100_2$.

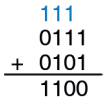


Рис. 1.9 Пример двоичного сложения

Цифровые системы обычно оперируют числами с заранее определенным и фиксированным количеством разрядов. Ситуацию, когда результат сложения превышает выделенное для него количество разрядов, называют *переполнением* (overflow). Четырехбитная ячейка памяти, например, может сохранять значения в диапазоне [0, 15]. Такая ячейка переполняется, если результат сложения превышает число 15. В этом случае дополнительный пятый бит отбрасывается, а результат, оставшийся в четырех битах, будет ошибочным. Переполнение можно обнаружить, если следить за переносом бита из наиболее значимого разряда двоичного числа (**рис. 1.8**), из наиболее левой колонки.

Пример 1.8 СЛОЖЕНИЕ С ПЕРЕПОЛНЕНИЕМ

Вычислить $1101_2 + 0101_2$. Будет ли переполнение?

Решение На **рис. 1.10** показано, что сумма равна 10010_2. Результат выходит за границы четырехбитового двоичного числа. Если его нужно запомнить в 4 битах, наиболее значимый бит пропадет, оставив некорректный результат 0010_2. Если вычисления производятся с числами с пятью или более битами, результат 10010_2 будет корректным.

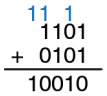


Рис. 1.10 Пример двоичного сложения с переполнением

1.4.6. Знак двоичных чисел

До сих пор мы рассматривали двоичные числа *без знака* (unsigned) – то есть только положительные числа. Часто для вычислений требуются как положительные, так и отрицательные числа, а это значит, что для знака двоичного числа нам потребуется дополнительный разряд. Существует

несколько способов представления двоичных чисел *со знаком* (signed). Наиболее широко применяются два: *прямой код* (Sign / Magnitude) и *дополнительный код* (Two's Complement).

Прямой код

Представление отрицательных двоичных с использованием прямого кода интуитивно покажется вам наиболее привлекательным, поскольку совпадает с привычным способом записи отрицательных чисел, когда сначала идет знак минус, а затем абсолютное значение числа. Двоичное число, состоящее из N бит и записанное в прямом коде, использует наиболее значимый бит для знака, а остальные $N - 1$ бит для записи абсолютного значения этого числа. Если наиболее значимый бит 0, то число положительное. Если наиболее значимый бит 1, то число отрицательное.

Пример 1.9 ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В ПРЯМОМ КОДЕ

Запишите числа 5 и −5 как четырехбитовые числа в прямом коде.

Решение Оба числа имеют абсолютную величину $5_{10} = 101_2$. Таким образом, $5_{10} = 0101_2$ и $-5_{10} = 1101_2$.

К сожалению, стандартный способ сложения не работает в случае двоичных чисел со знаком, записанных в прямом коде. Например, складывая $-5_{10} + 5_{10}$ привычным способом, получаем $1101_2 + 0101_2 = 10010_2$. Что, естественно, является полным абсурдом.

Двоичная переменная длиной N бит в прямом коде может представлять число в диапазоне $[-2^{N-1} + 1, 2^{N-1} - 1]$.

Другой несколько странной особенностью прямого кода является наличие +0 и −0, причем оба этих числа соответствуют одному нулю. Нетрудно предположить, что представление одной и той же величины двумя различными способами чревато ошибками.

Дополнительный код

Двоичные числа, записанные с использованием дополнительного кода, и двоичные числа без знака идентичны, за исключением того, что в случае дополнительного кода вес наиболее значимого бита -2^{N-1} вместо 2^{N-1}, как в случае двоичного числа без знака. Дополнительный код гарантирует однозначное представление нуля, допускает сложение чисел по привычной схеме, а значит, избавлен от недостатков прямого кода.

В случае дополнительного кода нулевое значение представлено нулями во всех разрядах двоичного числа: $00...000_2$. Максимальное положительное значение представлено нулем в наиболее значимом разряде и единицами во всех других разрядах двоичного числа: $01...111_2 = 2^{N-1} - 1$. Максимальное отрицательное значение содержит единицу в наиболее

значимом разряде и нули во всех остальных разрядах: $10...000_2 = -2^{N-1}$. Отрицательная единица представлена единицами во всех разрядах двоичного числа: $11...111_2$.

Обратите внимание на то, что наиболее значимый разряд у всех положительных чисел – это «0», в то время как у отрицательных чисел – это «1», то есть наиболее значимый бит дополнительного кода можно рассматривать как аналог знакового бита прямого кода. Но на этом сходство кончается, поскольку остальные биты дополнительного кода интерпретируются не так, как биты прямого кода.

В случае дополнительного кода знак отрицательного двоичного числа изменяется на противоположный путем выполнения специальной операции, называемой *в дополнительном коде* (taking the two's complement). Суть этой операции заключается в том, что инвертируются все биты этого числа, а затем к значению наименее значимого бита прибавляется 1. Подобная операция позволяет найти двоичное представление отрицательного числа или определить его абсолютное значение.

Пример 1.10 ПРЕДСТАВЛЕНИЕ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Найти представление -2_{10} как 4-битового числа в дополнительном коде.

Решение Начните с $+2_{10} = 0010_2$. Для получения -2_{10} инвертируйте биты и добавьте единицу. Инвертируя 0010_2, получим 1101_2. $1101_2 + 1 = 1110_2$. Итак, -2_{10} равно 1110_2.

Пример 1.11 ЗНАЧЕНИЕ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Найти десятичное значение числа 1001_2 в дополнительном коде.

Решение Число 1001_2 имеет старшую 1, поэтому оно должно быть отрицательным. Чтобы найти его модуль, инвертируем все биты и добавляем 1. Инвертируя 1001_2, получим 0110_2. $0110_2 + 1 = 0111_2 = 7_{10}$. Отсюда $1001_2 = -7_{10}$.

Неоспоримым преимуществом дополнительного кода является то, что привычный способ сложения работает как в случае положительных, так и отрицательных чисел. Напомним, что при сложении N-битных чисел N-й бит (т. е. $N + 1$ бит результата) не переносится.

(Фото: ESA/CNES/ ARIANESPACE-Service Optique CS6)

Ракета Ариан-5 ценой 7 млрд долларов, запущенная 4 июня 1996 года, отклонилась от курса и разрушилась через 40 секунд после запуска. Отказ был вызван тем, что в бортовом компьютере произошло переполнение 16-разрядных регистров, после чего компьютер вышел из строя.

Программное обеспечение Ариан-5 было тщательно протестировано, но на ракете Ариан-4. Но новая ракета имела двигатели с более высокими скоростными параметрами, которые, будучи переданными бортовому компьютеру, и вызвали переполнение регистров.

Пример 1.12 СЛОЖЕНИЕ ЧИСЕЛ, ПРЕДСТАВЛЕННЫХ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Вычислить (a) $-2_{10} + 1_{10}$ и (b) $-7_{10} + 7_{10}$ с помощью чисел в дополнительном коде.

Решение

(a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$.

(b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. Пятый бит отбрасывается, оставляя правильный 4-битовый результат 0000_2.

Вычитание одного двоичного числа из другого осуществляется путем преобразования вычитаемого в дополнительный код и последующего его сложения с уменьшаемым.

Пример 1.13 ВЫЧИТАНИЕ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Вычислить (a) $5_{10} - 3_{10}$ и (b) $3_{10} - 5_{10}$, используя 4-разрядные числа в дополнительном коде.

Решение

(a) $3_{10} = 0011_2$. Вычисляя его дополнительный код, получим $-3_{10} = 1101_2$. Теперь сложим $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Отметим, что перенос из наиболее значимой позиции сбрасывается, поскольку результат записывается в четырех битах.

(b) Вычисляя дополнительный код от 5_{10}, получим $-5_{10} = 1011_2$. Теперь сложим $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

Представление нуля в дополнительном коде также производится путем инвертирования всех битов (это дает $11...111_2$) и последующим прибавлением 1, что делает значения всех битов равными 0. При этом перенос наиболее значимого бита игнорируется. В результате нулевое значение всегда представлено набором только нулевых битов. В отличие от прямого кода дополнительный код не имеет отрицательного нуля. Ноль всегда считается положительным числом, так как его знаковый бит всегда 0.

Так же, как и двоичное число без знака, произвольное N-битное число, записанное в дополнительном коде, может принимать одно из 2^N возможных значений. Но весь этот диапазон разделен между положительным и отрицательным числами. Например, 4-битное двоичное число без знака может принимать 16 значений от 0 до 15. В случае дополнительного кода 4-битное число также принимает 16 значений, но уже от -8 до 7. В общем случае диапазон N-битного числа, записанного в дополнительном коде, охватывает $[-2^{N-1}, 2^{N-1} - 1]$. Легко понять, почему в отрицательном диапазоне оказалось на одно значение больше, чем в положительном, – в дополнительном коде отсутствует отрицательный ноль.

Максимальное отрицательное число, которое можно записать, используя дополнительный код $10...000_2 = -2^{N-1}$, иногда называют *странным числом* (weird number). Чтобы представить это число в дополнительном коде, инвертируем все его биты (это даст нам $01...111_2$), прибавим 1 и получим в результате $10...000_2$ – опять это же самое «странное» число. То есть это единственное отрицательное число, которое не имеет положительной пары.

В случае дополнительного кода сложение двух положительных или отрицательных N-битовых чисел может привести к переполнению, если результат будет больше, чем $2^{N-1} - 1$, или меньше, чем -2^{N-1}. Сложение положительного и отрицательного чисел, напротив, никогда не приводит к переполнению. В отличие от двоичного числа без знака перенос наиболее значимого бита не является признаком переполнения. Вместо этого индикатором переполнения является ситуация, когда после сложения двух чисел с одинаковым знаком знаковый бит суммы не совпадает со знаковыми битами слагаемых.

Пример 1.14 СЛОЖЕНИЕ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ С ПЕРЕПОЛНЕНИЕМ

Вычислить $4_{10} + 5_{10}$, используя четырехбитные числа в дополнительном коде. Произойдет ли переполнение?

Решение $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. Результат не помещается в диапазон положительных четырехбитных чисел в дополнительном коде, оказываясь отрицательным. Если бы вычисление выполнялось с пятью или более битами, результат был бы $01001_2 = 9_{10}$, что правильно.

В случае необходимости увеличения количества битов произвольного числа, записанного в дополнительном коде, значение знакового бита должно быть скопировано в наиболее значимые разряды модифицированного числа. Эта операция называется *знаковым расширением* (sign extension). Например, числа 3 и –3 записываются в 4-битном дополнительном коде как 0011 и 1101 соответственно. Если мы увеличиваем число разрядов до семи бит, мы должны скопировать знаковый бит в три наиболее значимых бита модифицированного числа, что дает 0000011 и 1111101.

Сравнение способов представления двоичных чисел

Три наиболее часто использующихся на практике способа представления двоичных чисел – это двоичные числа без знака, прямой код и дополнительный код. **Таблица 1.3** сравнивает диапазон N-битных чисел для каждого из этих трех способов. Преимущества дополнительного кода заключаются в том, что его можно использовать для представления

как положительных, так и отрицательных целых чисел, а привычный способ сложения работает для всех чисел, представленных в дополнительном коде. Вычитание осуществляется путем преобразования вычитаемого в отрицательное число и последующего сложения с уменьшаемым. В дальнейшем в этой книге, если не указано иное, предполагается, что все двоичные числа представлены в дополнительном коде.

Таблица 1.3 Диапазон *N*-битных чисел

Система	Диапазон
Двоичные числа без знака	$[0, 2^N - 1]$
Прямой код	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Дополнительный код	$[-2^{N-1}, 2^{N-1} - 1]$

На **рис. 1.11** изображена десятичная числовая шкала с соответствующими десятичными и 4-битными двоичными числами, представленными тремя вышеперечисленными способами. Двоичные числа без знака находятся в диапазоне [0, 15] и располагаются в обычном порядке. 4-битные двоичные числа, представленные в дополнительном коде, занимают диапазон [−8, 7]. Причем положительные числа [0, 7] используют точно такую же кодировку, как и двоичные числа без знака. Отрицательные же числа [−8, −1] кодируются таким образом, что наибольшее двоичное значение каждого такого числа без знака представляет число, наиболее близкое к 0. Обратите внимание на то, что «странное число» 1000 соответствует десятичному значению −8 и не имеет положительной пары. Числа, представленные в прямом коде, занимают диапазон [−7, 7]. При этом наиболее значимый бит является знаковым. Положительные числа [0, 7] используют такую же кодировку, как и двоичные числа без знака. Отрицательные числа симметричны положительным, с той лишь разницей, что их знаковый бит имеет значение 1. Ноль представлен двумя значениями 0000 и 1000. В результате того, что два числа соответствуют одному нулю, любое произвольное N-разрядное двоичное число в прямом коде может представлять только $2^N - 1$ целых числа.

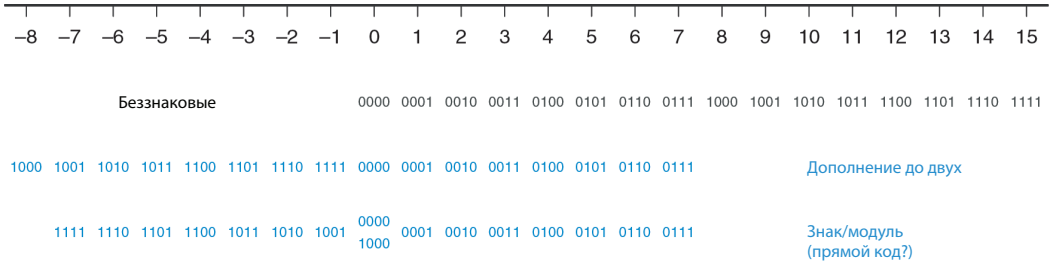


Рис. 1.11 Числовая шкала и 4-битовое двоичное кодирование

1.5. Логические элементы

Теперь, когда мы знаем, как использовать бинарные переменные для представления информации, рассмотрим цифровые системы, способные выполнять различные операции с этими переменными. *Логические элементы* (logic gates) – это простейшие цифровые схемы, получающие один или более двоичных сигналов на входе и производящие новый двоичный сигнал на выходе. При графическом изображении логических элементов для обозначения одного или нескольких входных сигналов и выходного сигнала используются специальные символы. Если смотреть на изображение логического элемента, то входные сигналы обычно размещаются слева (или сверху), а выходные сигналы – справа (или снизу). Разработчики цифровых систем обычно используют первые буквы латинского алфавита для обозначения входных сигналов и латинскую букву Y для обозначения выходного сигнала. Взаимосвязь между входными сигналами и выходным сигналом логического элемента может быть описана с помощью *таблицы истинности* (truth table) или логической функцией. Слева в таблице истинности представлены значения входных сигналов, а справа – значение соответствующего выходного сигнала. Каждая строка в такой таблице соответствует одной из возможных комбинаций входных сигналов. Логическая функция – это математическое выражение, описывающее логический элемент с помощью двоичных переменных.

1.5.1. Логический элемент НЕ

Логический *элемент НЕ* (NOT gate) имеет один вход A и один выход Y, как показано на **рис. 1.12**. Причем выходной сигнал Y – это сигнал, обратный входному сигналу A, или, как еще говорят, *инвертированный* A (inversed A). Если сигнал на входе A – это ЛОЖЬ, то сигнал на выходе Y будет ИСТИНА. Таблица истинности и *логическая функция* на **рис. 1.12** суммируют эту связь входного и выходного сигналов. В уравнении булевой логики линия над обозначением сигнала читается как «не», то есть математическое выражение $Y = \overline{A}$ произносится как «Y равняется не A». Именно поэтому логический элемент НЕ также называют *инвертором* (inverter).

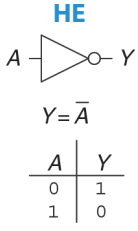
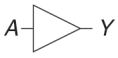


Рис. 1.12 Логический элемент НЕ

Для обозначения логического элемента НЕ используют и другие способы записи, включая $Y = A'$, $Y = \neg A$, $Y = !A$ и $Y = \sim A$. В этой книге мы будем пользоваться исключительно записью $Y = \overline{A}$, но не удивляйтесь, если в научной и технической литературе вы столкнетесь и с другими обозначениями.

БУФЕР



$Y = A$

A	Y
0	0
1	1

Рис. 1.13
Буфер

1.5.2. Буфер

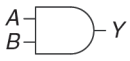
Другим примером логического элемента с одним входом является *буфер* (buffer), показанный на **рис. 1.13.**

Буфер просто копирует входной сигнал на выход. Если рассматривать буфер как часть логической схемы, то такой элемент ничем не отличается от простого провода и может показаться бесполезным. Вместе с тем на аналоговом уровне буфер может обеспечить характеристики, необходимые для нормального функционирования разрабатываемого устройства.

Буфер, например, необходим для передачи большого тока электродвигателю или для быстрой передачи сигнала сразу нескольким логическим элементам. Это еще один пример, доказывающий необходимость рассмотрения любой системы на нескольких уровнях абстракции, если мы хотим в полной мере понять эту систему. Рассмотрение буфера только с позиции цифрового уровня абстракции не позволяет нам разглядеть его реальное предназначение.

В логических схемах буфер обозначается треугольником. Кружок на выходе логического элемента в англоязычной литературе часто называемый *пузырем* (bubble), указывает на инверсию сигнала, как, например, показано на **рис. 1.12**.

И



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

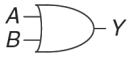
Рис. 1.14
Логический элемент И

1.5.3. Логический элемент И

Логические элементы с двумя входными сигналами гораздо интереснее, чем логический элемент НЕ и буфер. Логический *элемент И* (AND gate), приведенный на **рис. 1.14**, выдает значение ИСТИНА на выходе Y, исключительно только если оба входных сигнала A и B имеют значение ИСТИНА. В противном случае выходной сигнал Y имеет значение ЛОЖЬ.

В используемом нами соглашении входные сигналы перечислены в порядке 00, 01, 10, 11, как в случае подсчета в двоичной системе счисления. Логическая функция для логического элемента И может быть записано несколькими способами: $Y = A \cdot B$, $Y = AB$ или $Y = A \cap B$. Символ $\cap$ читается как «пересечение» и больше других нравится специалистам в математической логике. Но в этой книге мы предпочитаем использовать выражение $Y = AB$, которое звучит как «Y равно A и B», просто потому, что мы достаточно ленивы, чтобы выбрать то, что короче.

ИЛИ



$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Рис. 1.15
Логический элемент ИЛИ

1.5.4. Логический элемент ИЛИ

Логический *элемент ИЛИ* (OR gate), показанный на **рис. 1.15**, выдает значение ИСТИНА на выходе Y, если хотя бы один из двух входных сигналов A или B имеет значение ИСТИНА. Логическая функция для

элемента ИЛИ записывается как $Y = A + B$, или $Y = A$, или $Y = A \cup B$.

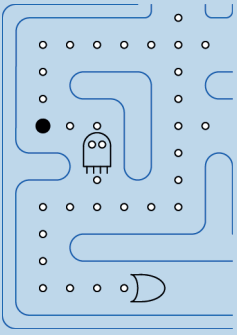
Символ $\cup$ читается как «объединение» и опять же больше всего нравится математикам. Разработчики цифровых систем обычно пользуются простым символом +. Математическое выражение $Y = A + B$ звучит «Y равно A или B».

По словам создателя языка программирования Perl Ларри Уолла, три основных достоинства программиста — это лень, нетерпение и самоуверенность.

1.5.5. Другие логические элементы с двумя входными сигналами

На **рис. 1.16** показаны другие широко распространенные логические элементы с двумя входными сигналами. Добавление кружка на выходе любого логического элемента превращает этот логический элемент в ему противоположный — то есть инвертирует его. Таким образом, например, из логического элемента И получается *логический элемент И-НЕ* (NAND gate). Значение выходного сигнала Y логического элемента И-НЕ будет ИСТИНА до тех пор, пока оба входных сигнала A и B не примут значение ИСТИНА. Точно так же из логического элемента ИЛИ получается *логический элемент ИЛИ-НЕ* (NOR gate). Его выходной сигнал Y будет ИСТИНА в том случае, если ни один из входных сигналов, ни A и ни B, не имеет значение ИСТИНА. *Исключающее ИЛИ с количеством входов, равным N* (N-input XOR gate), иногда еще называют элементом контроля по четности (parity gate). Такой логический элемент выдает на выход сигнал ИСТИНА, если нечетное количество входных сигналов имеет значение ИСТИНА. Как и в случае элемента с двумя входными сигналами, комбинации сигналов для элемента с N входами перечислены в логической таблице в порядке подсчета в двоичной системе счисления.

Забавный способ запомнить, как обозначается элемент ИЛИ на логических схемах, заключается в том, что графический символ ИЛИ напоминает главный персонаж компьютерной игры Pacman. Причем широко раскрытая пасть «голодного» ИЛИ находится со стороны входных сигналов и готова проглотить все сигналы ИСТИНА, которые только может найти!



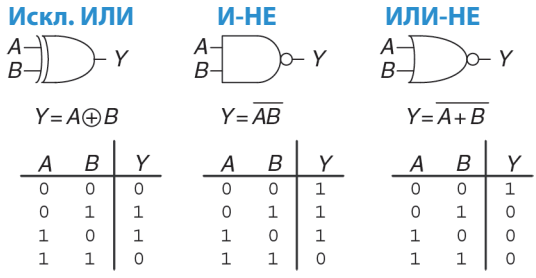
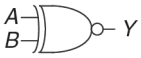


Рис. 1.16 Другие логические элементы с двумя входными сигналами

Искл. ИЛИ-НЕ



$Y = \overline{A \oplus B}$

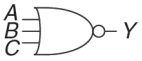
A	B	Y
0	0	
0	1	
1	0	
1	1	

Рис. 1.17 Логический элемент исключающее ИЛИ-НЕ

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Рис. 1.18 Таблица истинности логического элемента исключающее ИЛИ-НЕ

ИЛИ-НЕ



$Y = \overline{A + B + C}$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Рис. 1.19 Логический элемент ИЛИ-НЕ с тремя входами

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Рис. 1.20 Таблица истинности логического элемента ИЛИ-НЕ с тремя входами

Пример 1.15 ЛОГИЧЕСКИЙ ЭЛЕМЕНТ ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ

На **рис. 1.17** показаны обозначение и логическая функция элемента Исключающее ИЛИ-НЕ (XNOR) с двумя входами, который выполняет инверсию Исключающего ИЛИ. Заполните таблицу истинности.

Решение На **рис. 1.18** представлена таблица истинности. Выход Исключающего ИЛИ-НЕ есть ИСТИНА, если оба входа имеют значение ЛОЖЬ или оба входа имеют значение ИСТИНА.

Логический элемент Исключающее ИЛИ-НЕ с двумя входами иногда называют логическим элементом равенства, так как его выход есть ИСТИНА, когда входы совпадают.

1.5.6. Логические элементы с количеством входов больше двух

Многие логические функции, а значит, и логические элементы, необходимые для их реализации, оперируют тремя и более входными сигналами. Наиболее распространенные из таких логических элементов – это И, ИЛИ, Исключающее ИЛИ, И-НЕ, ИЛИ-НЕ и Исключающее ИЛИ-НЕ. Логический элемент И с количеством входов, равным N, выдает значение ИСТИНА, когда значения на всех N входах этого логического элемента ИСТИНА. Логический элемент ИЛИ с количеством входов, равным N, выдает ИСТИНА, когда значение хотя бы одного из его входов ИСТИНА.

Пример 1.16 ЛОГИЧЕСКИЙ ЭЛЕМЕНТ ИЛИ-НЕ С ТРЕМЯ ВХОДАМИ

На **рис. 1.19** показаны обозначение и логическая функция для логического элемента ИЛИ-НЕ с тремя входами. Заполните таблицу истинности.

Решение На **рис. 1.20** показана таблица истинности. Выход есть ИСТИНА, только если нет ни одного входа со значением ИСТИНА.

Пример 1.17 ЛОГИЧЕСКИЙ ЭЛЕМЕНТ И С ЧЕТЫРЬМЯ ВХОДАМИ

На **рис. 1.21** показаны обозначение и логическая функция для логического элемента И с четырьмя входами. Заполните таблицу истинности.

Решение На **рис. 1.22** показана таблица истинности. Выход есть ИСТИНА, только если все входы имеют значение ИСТИНА.

1.6. За пределами цифровой абстракции

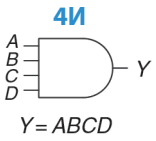


Рис. 1.21 **Логический элемент И с четырьмя входами**

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рис. 1.22 **Таблица истинности логического элемента И с четырьмя входами**

Цифровая система оперирует дискретными переменными. Но для представления этих переменных используются непрерывные физические величины, такие как напряжение в электрической цепи, положение шестеренок в механической передаче или уровень жидкости в гидравлическом цилиндре. Задача разработчика цифровой системы – определить, каким образом непрерывно меняющаяся величина соотносится с конкретным значением дискретной переменной.

Рассмотрим, например, задачу представления двоичного сигнала A напряжением в электрической цепи. Допустим, что напряжение 0 В соответствует значению $A = 0$, а напряжение 5 В соответствует $A = 1$. Но реальная цифровая система должна быть устойчива к неизбежному в такой ситуации шуму, так что значение 4,97 В, вероятно, также следует толковать как $A = 1$. А что делать, если напряжение равно 4,3 В? Или 2,8 В? Или 2,500000 В?

1.6.1. Напряжение питания

Предположим, что минимальное напряжение в электронной цифровой системе, называемое также *напряжением земли* (ground voltage, или просто ground, GND), составляет 0 В. Самое высокое напряжение в системе поступает от блока питания и, как правило, обозначается V_{DD}. Транзисторные технологии семидесятых и восьмидесятых годов прошлого века в основном использовали V_{DD}, равное 5 В. С переходом на транзисторы меньшего размера V_{DD} последовательно снижали до 3,3 В, 2,5 В, 1,8 В, 1,5 В, 1,2 В и даже ниже для экономии электроэнергии и для избежания перегрузки транзисторов.

1.6.2. Логические уровни

Отображение непрерывно меняющейся переменной на различные значения дискретной двоичной переменной выполняется путем определения *логических уровней*, как показано на **рис. 1.23**. Первый логический элемент в рассматриваемой схеме называется *источник* (driver), а второй – *приемник* (receiver). Выходной сигнал источника подключается ко входу приемника. Источник выдает выходной сигнал низкого напряжения (0) в диапазоне от 0 В до V_{OL} или выходной сигнал высокого напряжения (1) в диапазоне от V_{OH} до V_{DD}. Если приемник получает на вход сигнал в диапазоне от 0 до V_{IL}, он рассматривает такой сигнал как ноль. Если

V_{DD} обозначает напряжение *стока* (drain) в транзисторах, построенных на структуре металл-оксид-полупроводник (МОП). Такие транзисторы используются сегодня для создания самых современных микросхем. Напряжение источника питания иногда также обозначают V_{CC}, как напряжение *коллектора* (collector) в биполярных транзисторах более ранних микросхем. Напряжение земли (*ground voltage*, или просто *ground*) иногда обозначают как V_{SS} потому, что это напряжение на *истоке* (source) МОП-транзистора. Для более подробной информации о том, как функционирует транзистор, см. раздел 1.7.

приемник получает на вход сигнал в диапазоне от V_{IH} до V_{DD}, он рассматривает такой сигнал как единицу. Если же по какой-либо причине, например наличия шумов или неисправности одного из элементов схемы, напряжение сигнала на входе приемника падает настолько, что попадает в *запрещенную зону* (forbidden zone) между V_{IL} и V_{IH}, то поведение этого логического элемента становится непредсказуемым. V_{OH} и V_{OL} называются соответственно *высоким и низким логическими уровнями выхода* (output high and low logic levels), а V_{IH} и V_{IL} называются соответственно *высоким и низким логическими уровнями входа* (input high and low logic levels).

1.6.3. Допускаемые уровни шумов

Для того чтобы выходной сигнал источника был правильно интерпретирован на входе приемника, необходимо, чтобы $V_{OL} < V_{IL}$ и $V_{OH} > V_{IH}$. В этом случае, даже если выходной сигнал источника будет загрязнен шумами, приемник по-прежнему сможет правильно определить логический уровень входного сигнала. *Допускаемый уровень шумов* (noise margin) – это то максимальное количество шума, присутствие которого в выходном сигнале источника не мешает приемнику корректно интерпретировать значение полученного сигнала. Согласно **рис. 1.23**, значения *нижнего допускаемого уровня шумов* (low noise margin) и *верхнего допускаемого уровня шумов* (high noise margin) определяются следующим образом:

$$NM_L = V_{IL} - V_{OL} \quad (1.2)$$

$$NM_H = V_{OH} - V_{IH} \quad (1.3)$$

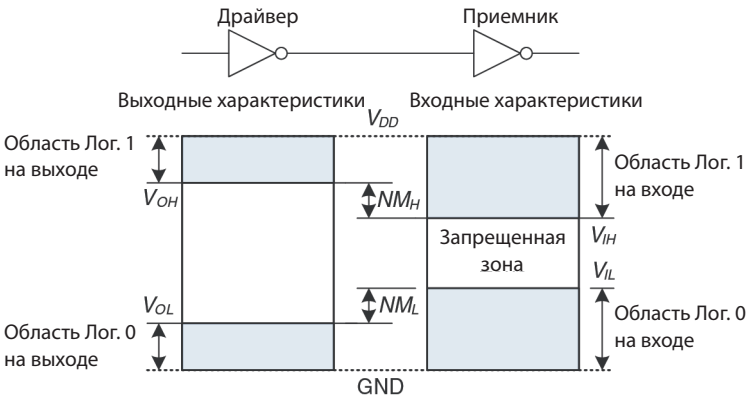


Рис. 1.23 Логические уровни и уровни шума

Пример 1.18 РАСЧЕТ УРОВНЕЙ ШУМА

Рассмотрим схему с инверторами на **рис. 1.24**. V_{O1} – это напряжение на выходе инвертора I1, а V_{I2} – напряжение на входе инвертора I2. Оба инвертора имеют следующие характеристики: V_{DD} = 5 В, V_{IL} = 1,35 В, V_{IH} = 3,15 В, V_{OL} = 0,33 В и V_{OH} = 3,84 В. Определите нижний и верхний уровни шума. Может ли схема корректно обработать уровень шума в 1 В между V_{O1} и V_{I2}?

Рис. 1.24 Схема с инверторами

Решение Границы уровня шума инвертора следующие: $NM_L = V_{IL} - V_{OL}$ = (1,35 В – 0,33 В) = 1,02 В, $NM_H = V_{OH} - V_{IH}$ = (3,84 В – 3,15 В) = 0,69 В. Схема может корректно обработать шум в 1 В, когда на выходе НИЗКИЙ уровень (NM_L = 1,02 В), но не когда на выходе ВЫСОКИЙ уровень (NM_H = 0,69 В). Например, предположим, что инвертор I1 имеет на выходе в наихудшем случае ВЫСОКОЕ значение, $V_{O1} = V_{OH}$ = 3,84 В. Если наличие шума вызовет падение напряжения на 1 В на входе инвертора I2, тогда V_{I2} = (3,84 В – 1 В) = 2,84 В. Это меньше, чем допустимое входное значение ВЫСОКОГО уровня, V_{IH} = 3,15 В, поэтому инвертор I2 может не принять правильное входное значение ВЫСОКОГО уровня.

1.6.4. Передаточная характеристика

Для понимания предела цифровой абстракции мы должны рассмотреть поведение логических элементов с аналоговой точки зрения. *Передаточная характеристика* (DC transfer characteristics) какого-либо логического элемента описывает напряжение на выходе этого элемента как функцию напряжения на его входе, когда входной сигнал изменяется настолько медленно, что выходной сигнал успевает изменяться вслед за ним. Такая характеристика называется передаточной, поскольку описывает взаимосвязь между входным и выходным напряжениями.

В случае идеального инвертора переключение будет резким в точке $V_{DD}/2$, как показано на **рис. 1.25 (a)**. Для $V(A) < V_{DD}/2$ $V(Y) = V_{DD}$. Для $V(A) > V_{DD}/2$ $V(Y) = 0$. В этом случае $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ и $V_{OL} = 0$.

Напряжение при переключении реального инвертора изменяется постепенно между граничными значениями, как показано на **рис. 1.25 (b)**. Если входное напряжение $V(A)$ равно 0, то напряжение на выходе $V(Y) = V_{DD}$. Если $V(A) = V_{DD}$, то $V(Y) = 0$. Но переход между этими конечными точками плавный и может находиться правее или

DC указывает на состояние, когда напряжение на входе электронной системы поддерживается постоянным или изменяется так медленно, что остальные параметры системы плавно изменяются вместе с ним. Исторически термин DC ведет свое происхождение от понятия постоянный ток (direct current) – метод передачи электрической энергии по схеме на расстояние, когда напряжение в линии поддерживается постоянным. В отличие от DC, переходная характеристика (transient response) схемы – это состояние, когда входное напряжение меняется быстро. Переходные процессы рассматриваются в разделе 2.9.

левее значения $V_{DD}/2$. В связи с этим возникает закономерный вопрос, как в этом случае определить логические уровни.

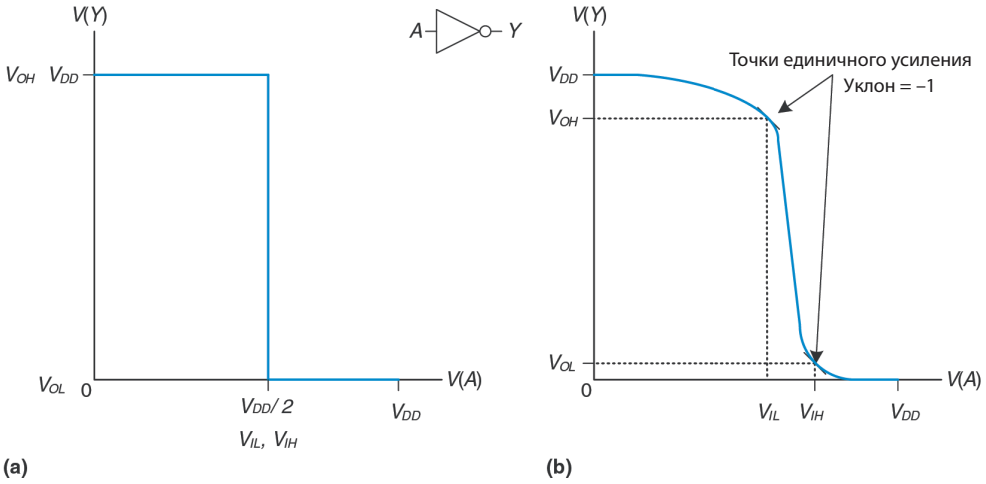


Рис. 1.25 Передаточные характеристики и уровни шума

Разумно выбрать в качестве логических уровней те две точки, где наклон передаточной характеристики $\mathrm{d}V(Y)/\mathrm{d}V(A)$ равен -1. Такие точки называются *граничные коэффициенты передачи* (unity gain points). Подобный выбор обычно максимизирует допускаемые уровни шумов. При уменьшении V_{IL} V_{OH} увеличивается незначительно. Но если V_{IL} растет, V_{OH} падает практически отвесно.

1.6.5. Статическая дисциплина

Для того чтобы избежать попадания входных сигналов в запретные зоны, логические элементы должны разрабатываться в соответствии с *принципом статической дисциплины* (static discipline). Принцип статической дисциплины требует, чтобы при условии наличия логически корректных сигналов на входе каждый элемент системы выдавал логически корректные сигналы на выходе.

Применение принципа статической дисциплины ограничивает свободу разработчика в выборе аналоговых элементов для построения цифровых систем, но помогает обеспечить простоту и надежность разрабатываемых цифровых схем. Используя этот принцип, разработчик поднимается с аналогового уровня абстракции на цифровой, что увеличивает производительность разработчика, избавляя его от рассмотрения излишних деталей.

Выбор V_{DD} и логических уровней может быть произвольным, но этот выбор должен обеспечить совместимость всех логических элементов, об-

менивающихся данными в пределах одной цифровой системы. Поэтому логические элементы обычно группируются в *семейства логики* (logic families) таким образом, что любой элемент из одного семейства при соединении с любым другим элементом из этого же семейства автоматически обеспечивает соблюдение принципа статической дисциплины. Логические элементы одного семейства соединяются друг с другом так же легко, как и блоки конструктора Lego, поскольку они полностью совместимы по напряжению источника питания и логическим уровням.

Четыре основных семейства логических элементов доминировали с 70-х по 90-е годы прошлого века – это *ТТЛ – транзисторно-транзисторная логика* (Transistor-Transistor Logic, или TTL), *КМОП – логика, построенная на комплементарной структуре металл-оксид-полупроводник* (Complementary Metal-Oxide-Semiconductor Logic, или CMOS), *НТТЛ – низковольтная транзисторно-транзисторная логика* (Low-Voltage Transistor-Transistor Logic, или LVTTL) и *НКМОП – низковольтная логика на комплементарной структуре металл-оксид-полупроводник* (Low-Voltage Complementary Metal-Oxide-Semiconductor Logic, или LVCMOS). Логические уровни для всех этих семейств представлены в **табл. 1.4**. Начиная с 90-х годов прошлого века четыре вышеперечисленных семейства распались на большое количество более мелких семейств в связи со все большим распространением устройств, требующих еще более низкого напряжения питания. В **приложении А.6** наиболее распространенные семейства логических элементов рассматриваются детально.

Таблица 1.4 Семейства логики с уровнями напряжения 5 В и 3,3 В

Семейство логики	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4,75–5,25)	0,8	2,0	0,4	2,4
CMOS	5 (4,5–6)	1,35	3,15	0,33	3,84
LVTTL	3,3 (3–3,6)	0,8	2,0	0,4	2,4
LVCMOS	3,3 (3–3,6)	0,9	1,8	0,36	2,7

Пример 1.19 СОВМЕСТИМОСТЬ ЛОГИЧЕСКИХ СЕМЕЙСТВ

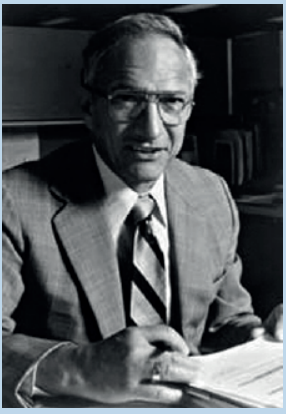
Какие из логических семейств из **табл. 1.4** могут надежно взаимодействовать между собой?

Решение В **табл. 1.5** перечислены логические семейства, которые имеют совместимые логические уровни. Заметим, что пятивольтовые логические семейства, такие как TTL и CMOS, могут выдавать на выход ВЫСОКИЙ уровень в 5 В. Если этот пятивольтовый сигнал подается на вход семейству с уровнем 3,3 В, такому как LVTTL или LVCMOS, это может повредить приемник, если в спецификации последнего не указано прямо, что он «5 В-совместимый».

Таблица 1.5 Совместимость логических семейств

		Приемник			
		TTL	CMOS	LVTTL	LVCMOS
Источник	TTL	Да	Нет: $V_{OH} < V_{IH}$	Возможно*	Возможно*
	CMOS	Да	Да	Возможно*	Возможно*
	LVTTL	Да	Нет: $V_{OH} < V_{IH}$	Да	Да
	LVCMOS	Да	Нет: $V_{OH} < V_{IH}$	Да	Да

* Если сигнал в 5 В ВЫСОКОГО уровня не может повредить вход приемника.



Роберт Нойс
1927–1990

Родился в городе Берлингтон штата Айова и получил степень бакалавра в области физики в Гриннеллском колледже, а степень доктора наук в области физики — в Массачусетском технологическом институте. Роберта Нойса прозвали «мэром Силиконовой долины» за его обширный вклад в развитие микроэлектроники.
Нойс стал сооснователем Fairchild Semiconductor в 1957 году и корпорации Intel в 1968 году. Он также является одним из изобретателей интегральной микросхемы. Инженеры из групп, возглавляемых Нойсом, в дальнейшем основали целый ряд выдающихся полупроводниковых компаний. (Воспроизводится с разрешения Intel Corporation © 2006 г.)

1.7. КМОП-транзисторы

Аналитическая машина Бэббиджа была механическим устройством с пружинами и шестеренками, а в первых компьютерах использовались реле или вакуумные трубки. Современные компьютеры используют транзисторы, потому что они дешевы, имеют небольшие размеры и высокую надежность. Транзистор – это переключатель с двумя положениями «включить» и «выключить», контролируемый путем подачи напряжения или тока на управляющую клемму. Существуют два основных типа транзисторов – *биполярные транзисторы* (bipolar junction transistors) и *МОП-транзисторы* (металл-оксид-полупроводник-транзисторы), иногда говорят *полевые транзисторы* (metal-oxide-semiconductor field effect transistors, или MOSFET).

В 1958 году Джек Килби из Texas Instruments создал первую интегральную схему, состоявшую из двух транзисторов. В 1959 году Роберт Нойс, работавший тогда в Fairchild Semiconductor, запатентовал метод соединения нескольких транзисторов на одном кремниевом чипе. В то время один транзистор стоил около 10 американских долларов.

Сегодня, после более чем трех десятилетий беспрецедентного развития полупроводниковой технологии, инженеры могут «упаковать» приблизительно один миллиард полевых МОП-транзисторов на одном квадратном сантиметре кремниевого чипа, причем каждый из этих транзисторов будет стоить меньше десяти микроцентов. Плотность размещения транзисторов на чи-

пе возрастает, а себестоимость одного транзистора снижается на порядок каждые восемь лет.

В настоящее время полевые МОП-транзисторы – это те «кирпичики», из которых собираются почти все цифровые системы. В этом разделе мы выйдем за пределы цифровой абстракции и внимательно рассмотрим, как можно построить логические элементы из полевых МОП-транзисторов.

1.7.1. Полупроводники

МОП-транзисторы изготовляются из кремния – элемента, преобладающего в скальной породе и песке. Кремний (Si) – это элемент IV атомной группы, то есть он имеет четыре валентных электрона, может образовывать связи с четырьмя соседними атомами и, таким образом, формировать кристаллическую *решетку* (lattice). На **рис. 1.26 (a)**, для простоты, кристаллическая решетка показана в двумерной системе координат, при этом полезно помнить, что реальная кристаллическая решетка имеет форму куба. Линия на **рис. 1.26 (a)** изображает ковалентную связь. По своей природе кремний – плохой проводник, потому что все электроны заняты в ковалентных связях. Но проводимость кремния улучшается, если добавить в него небольшое количество атомов другого вещества, называемого *примесью* (dopant). Если в качестве примеси используется элемент V атомной группы, например мышьяк (As), то в каждом атоме примеси окажется дополнительный электрон, не участвующий в образовании ковалентных связей. Этот свободный электрон может легко перемещаться внутри кристаллической решетки. При этом атом мышьяка, потерявший электрон, превращается в положительный ион (As^+), как показано на **рис. 1.26 (b)**. Электрон имеет *отрицательный заряд* (negative charge), поэтому мышьяк принято называть примесью *n-типа* (*n*-type dopant). Если же в качестве примеси используется элемент III атомной группы, например бор (B), то в каждом из атомов примеси будет не хватать одного электрона, как показано на **рис. 1.26 (c)**. Отсутствующий электрон называют *дыркой* (hole). Электрон из соседнего атома кремния может перейти к атому бора и заполнить недостающую связь. При этом атом бора, получивший дополнительный электрон, превращается в отрицательный ион (B^-), а в атоме кремния возникает дырка. Таким образом, дырка может мигрировать в кристаллической решетке подобно электрону. Дырка – это всего лишь отсутствие отрицательного заряда, но она ведет себя в полупроводнике как положительно заряженная частица. Именно поэтому бор называют примесью *p-типа* (*p*-type dopant). А поскольку проводимость кремния может меняться на порядки в зависимости от концентрации примесей, кремний называют *полупроводником* (semiconductor).

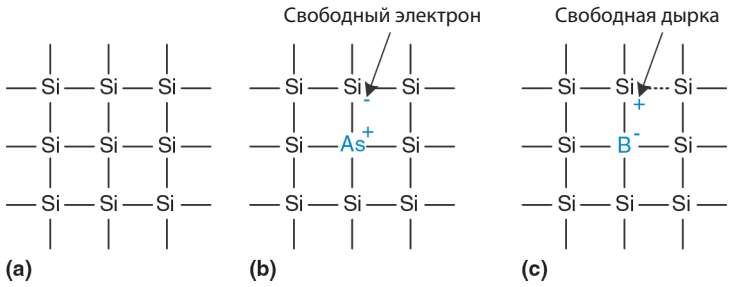


Рис. 1.26 Кремниевая решетка и атомы примесей

1.7.2. Диоды

Диод (diode) – это соединение полупроводника *p*-типа с полупроводником *n*-типа, как показано на **рис. 1.27**. При этом область *p*-типа называют *анодом* (anode), а область *n*-типа – *катодом* (cathode). Когда напряжение на аноде превышает напряжение на катоде, *диод открыт* (forward biased), и ток через него течет от анода к катоду. Если же напряжение на аноде ниже напряжения на катоде, то диод *закрыт* (reverse biased), и ток через диод не течет. Символ диода очень интуитивен и наглядно показывает, что ток через диод может протекать только в одном направлении.

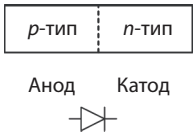


Рис. 1.27 Структура диода с *p-n*-соединением и его обозначение

1.7.3. Конденсаторы

Конденсатор (capacitor) состоит из двух проводников, отделенных друг от друга изолятором. Если к одному из проводников приложить напряжение V, то через некоторое время этот проводник накопит электрический заряд Q, а другой проводник накопит противоположный электрический заряд $-Q$. *Емкостью* (capacitance) конденсатора C называется отношение заряда к приложенному напряжению $C = Q/V$. Емкость прямо пропорциональна размеру проводников и обратно пропорциональна расстоянию между ними. Символ, используемый для обозначения конденсатора, показан на **рис. 1.28**.

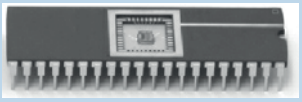
Рис. 1.28 Обозначение конденсатора

Емкость – это очень важный параметр электрической схемы, поскольку зарядка или разрядка любого проводника требует времени и энергии. Более высокая емкость означает, что электрическая схема будет работать медленнее и потребует для своего функционирования больше энергии. К понятиям скорости и энергии мы будем постоянно возвращаться на протяжении всей этой книги.

1.7.4. *n*-МОП- и *p*-МОП-транзисторы

Полевой МОП-транзистор представляет собой «сэндвич» из нескольких слоев проводящих и изолирующих материалов. «Фундамент», с которого начинается построение полевых МОП-транзисторов, – это тонкая круглая кремниевая пластина (wafer) приблизительно от 15 до 30 см в диаметре, в русскоязычной литературе называемая подложкой, вафлей или вэйфером. Производственный процесс начинается с пустой подложки. Этот процесс включает заранее определенную последовательность операций, в ходе которых примеси имплантируются в кремний, на подложке выращиваются тонкие пленки кремния и диоксида кремния, и наносится слой металла. После каждой операции на подложку в качестве маски наносится определенный рисунок (pattern), чтобы наносимый в ходе следующей операции материал оставался лишь в тех местах, где он необходим. Поскольку размеры одного транзистора – это доли микрона[1], а вся подложка обрабатывается в ходе одного производственного процесса, когда одновременно производятся миллиарды транзисторов, себестоимость одного транзистора существенно снижается. После того как все операции завершены, подложка нарезается на прямоугольные пластины, называемые в англоязычной литературе *chip* (чип) или *dice*, причем на каждом из этих прямоугольников размещаются тысячи, миллионы или даже миллиарды транзисторов. Каждый такой чип тестируется, а затем помещается в пластиковый или керамический *корпус-упаковку* (package) с металлическими *контактами* (pins), для того чтобы его можно было установить на монтажной плате.

Технические специалисты компании Intel не могут войти в чистое помещение, где производятся микросхемы, без защитного комбинезона Gore-Tex, называемого на профессиональном сленге «костюмом кролика» (bunny suit). Наличие такого комбинезона предотвращает от загрязнения кремниевые подложки с микроскопическими транзисторами на них от частиц одежды, кожи или волос. (Воспроизводится с разрешения корпорации Intel©, 2006 г.)



Корпус с рядом выводов по обеим длинным сторонам (Dual-Inline Package, или DIP) с 40 металлическими контактами – по 20 на каждой стороне – содержит внутри небольшой чип (на рисунке он практически не виден). Этот чип соединяется с ножками контактов золотыми проводами, каждый из которых тоньше человеческого волоса.

(Фотография Кевина Марра. © Харви колледж)

Сэндвич полевого МОП-транзистора состоит из слоя проводника, называемого *затвором* (gate), наложенного на слой изолятора – диоксида кремния (SiO_2), в свою очередь наложенного на кремниевую пластину, называемую подложкой. Изначально для изготовления затвора использовался тонкий слой металла, отсюда и название этого типа транзисторов – металл-оксид-полупроводник. В современных же технологических процессах в качестве материала затвора используется поликристаллический кремний, поскольку кремний не плавится в ходе последующей высокотемпературной обработки кристалла. Диоксид кремния – это хорошо известное всем нам стекло, и в полупроводниковой промышленности этот материал часто

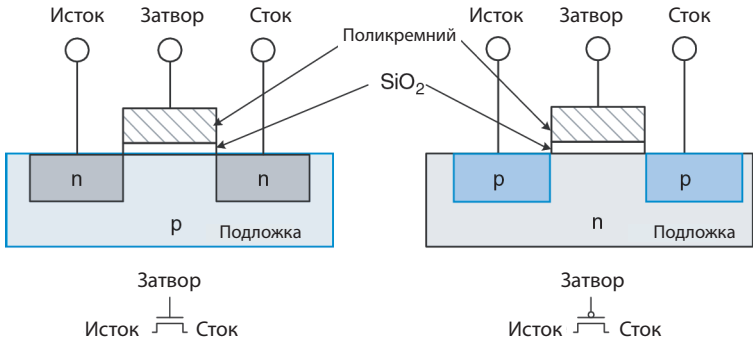
[1] 1 μm = 1 мкм = 10^{-6} м.

С физической точки зрения исток и сток симметричны. Вместе с тем мы будем говорить, что электрический заряд перетекает от истока к стоку. В *n*-МОП-транзисторе электрический заряд переносится электронами, которые двигаются из зоны с отрицательным напряжением в зону с положительным напряжением. В *p*-МОП-транзисторе заряд переносится дырками, которые двигаются из зоны с положительным напряжением в зону с отрицательным напряжением. Если схематически изобразить транзистор таким образом, чтобы зона максимального положительного напряжения находилась сверху, а зона максимального отрицательного напряжения снизу, то источником (отрицательного) заряда в *n*-МОП-транзисторе будет нижний вывод, а источником (положительного) заряда в *p*-МОП-транзисторе будет верхний вывод.

называют просто оксидом. Слои металл-оксид-полупроводника образуют конденсатор, в котором тонкий слой оксида (или окисла), называемого диэлектриком, изолирует металлическую пластину от полупроводниковой.

Существуют два вида полевых МОП-транзисторов: *n*-МОП и *p*-МОП (по английски *n*-MOS и *p*-MOS, что произносится как *н-мосс* и *пи-мосс*). На **рис. 1.29** схематически показано сечение каждого из этих двух типов транзисторов так, как будто мы распилили кристалл и теперь смотрим на транзистор сбоку. В транзисторах *n*-типа, называемых *n*-МОП, области, где расположены полупроводниковые примеси *n*-типа – в свою очередь называемые *истоком* (source) и *стоком* (drain), – находятся рядом с *затвором* (gate), причем вся эта структура размещается на подложке *p*-типа. В транзисторах же *p*-МОП и исток, и сток – это области *p*-типа, размещенные на подложке *n*-типа.

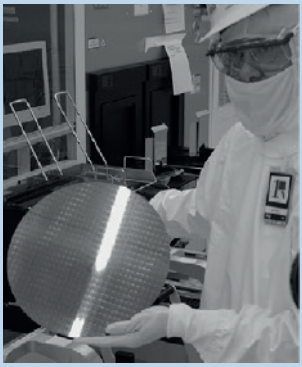
Полевой МОП-транзистор ведет себя как переключатель, управляемый приложенным к нему напряжением. В таком транзисторе напряжение перехода создает электрическое поле, включающее или выключающее линию связи между источником и стоком. Термин *полевой транзистор* (field effect transistor) является прямым отражением принципа работы такого устройства. Знакомство с работой полупроводниковых устройств мы начнем с изучения *n*-МОП-транзистора.



(a) *n*-МОП **(b)** *p*-МОП

Рис. 1.29 ***n*-МОП- и *p*-МОП-транзисторы**

Подложка *n*-МОП-транзистора обычно находится под напряжением земли GND, которое является минимальным напряжением в системе.



Технический специалист корпорации Intel держит в руках 12-дюймовый вейфер с несколькими сотнями микропроцессоров на нем. (Воспроизводится с разрешения корпорации Intel©, 2006 г.)

Для начала рассмотрим случай, когда, как показано на **рис. 1.30 (a)**, напряжение на затворе также равно 0 В. Диоды между истоком или стоком и подложкой находятся в состоянии, называемом *обратным смещением* (reverse bias), поскольку напряжение на истоке и стоке не является отрицательным. В результате этого канал для движения тока между истоком и стоком остается закрытым, а транзистор выключенным. Теперь рассмотрим ситуацию, когда напряжение на затворе повышается до V_{DD} – так, как показано на **рис. 1.30 (b)**. Если приложить положительное напряжение к затвору (верхней пластине конденсатора), то это создает электрическое поле между затвором и подложкой, в результате в зоне между истоком и стоком под слоем окисла формируется избыток электронов. При достаточно высоком напряжении на нижней границе затвора накапливается настолько много электронов, что область с полупроводником *p*-типа превращается в область с полупроводником *n*-типа. Такая инвертированная область называется *каналом* (channel). В этот момент в транзисторе образуется область проводимости от источника *n*-типа через каналы *n*-типа к стоку *n*-типа, и через этот канал электроны могут беспрепятственно перемещаться от истока к стоку. Транзистор включен. Напряжение перехода, которое требуется для включения транзистора, называется *пороговым значением напряжения* (threshold voltage) V_T и обычно составляет от 0,3 до 0,7 В.

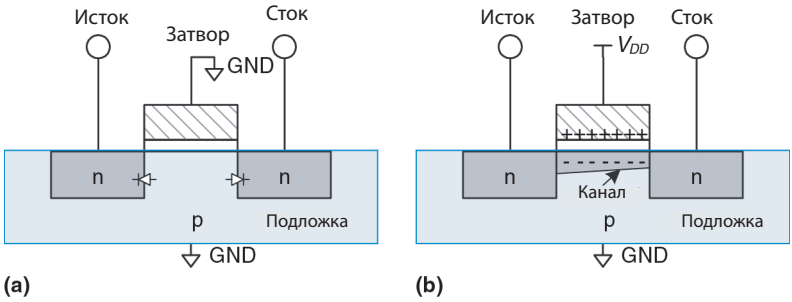
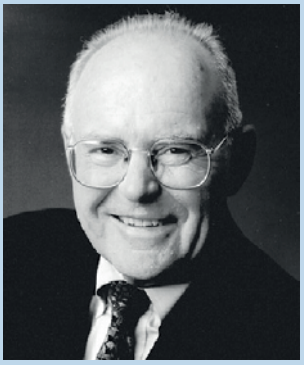


Рис. 1.30 Работа *n*-МОП-транзистора

Транзистор *p*-МОП работает с точностью до наоборот, как вы, возможно, уже догадались по наличию точки в обозначении этого типа транзистора на **рис. 1.31**. Подложка *p*-МОП-транзистора находится под напряжением V_{DD}. Если затвор также находится под напряжением V_{DD}, то *p*-МОП-транзистор выключен. Если же на затвор подается напряжение земли GND, проводимость канала инвертируется, превращаясь в проводимость *p*-типа, и транзистор включается.



Гордон Мур
1929–

Гордон Мур родился в Сан-Франциско. Мур получил степень бакалавра в области химии в университете штата Калифорния и степень доктора в области химии и физики в Калифорнийском Технологическом университете (Caltech). В 1968 году Гордон Мур и Роберт Нойс основали корпорацию Intel. В 1965 году Мур заметил, что полупроводниковые технологии развиваются с такой скоростью, что число транзисторов, которое можно разместить на одной микросхеме, удваивается каждый год.

Сегодня эта тенденция известна как закон Мура. Начиная с 1975 года количество транзисторов на одной микросхеме удваивается каждые два года. Одно из следствий закона Мура гласит, что производительность микропроцессоров удваивается за период от 18 до 24 месяцев. Продажи же полупроводниковых устройств растут по экспоненте. К сожалению, потребление электроэнергии также имеет тенденцию к экспоненциальному росту.

(Воспроизводится с разрешения корпорации Intel ©, 2006 г.)

К сожалению, полевые МОП-транзисторы в роли переключателя работают далеко не идеально. В частности, *n*-МОП-транзисторы хорошо передают 0, но плохо передают 1. Если переход *n*-МОП-транзистора находится под напряжением V_{DD}, то напряжение на стоке будет колебаться между 0 и $V_{DD} - V_T$. Аналогичным же образом *p*-МОП-транзисторы хорошо передают 1, но плохо передают 0. Но, как мы увидим в дальнейшем, возможно построить хорошо работающий логический элемент, используя только те режимы *n*-МОП- и *p*-МОП-транзисторов, в которых их работа близка к идеальной.

Для изготовления *n*-МОП-транзистора требуется подложка с проводимостью *p*-типа, а для изготовления *p*-МОП-транзисторов необходима подложка *n*-типа. Для того чтобы разместить оба типа транзисторов на одном чипе, производственный процесс, как правило, начинается с подложки *p*-типа, в который затем имплантируют области для размещения *p*-МОП-транзисторов *n*-типа, называемые *колодцами* (wells). Такой процесс называется *комплементарным МОП*, или *КМОП* (Complementary MOS, или CMOS). В настоящее время КМОП-процесс используется для изготовления подавляющего большинства транзисторов и микросхем.

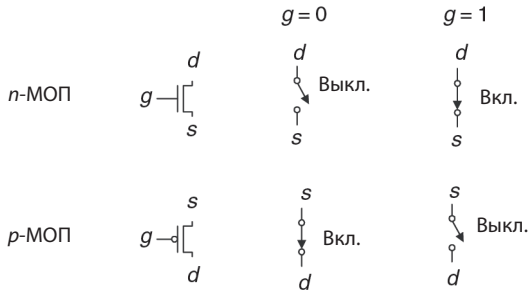


Рис. 1.31 Модели переключения полевых МОП-транзисторов

Подведем итог. КМОП-процесс позволяет разместить МОП-транзисторы *n*-типа и *p*-типа, показанные на **рис. 1.31**, на одном чипе. Напряжение на затворе (g) управляет током между истоком (s) и стоком (d). Транзисторы *n*-МОП выключены, когда значение напряжения на затворе соответствует логическому 0, и включены, когда значение напряжения на затворе соответствует логической 1. Транзисторы *p*-МОП, напротив, включены, когда значение напряжения на затво-

ре соответствует логическому 0, и выключены, когда значение напряжения на затворе соответствует логической 1.

1.7.5. Логический элемент НЕ на КМОП-транзисторах

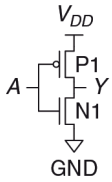


Рис. 1.32 Схема логического элемента НЕ

Схема на **рис. 1.32** демонстрирует, как можно построить логический элемент НЕ, используя КМОП-транзисторы.

На этой схеме треугольник обозначает напряжение земли GND, а горизонтальная линия обозначает напряжение питания V_{DD}. На всех последующих схемах в этой книге мы не будем использовать буквенные обозначения V_{DD} и GND. *n*-МОП-транзистор N1 включен между землей GND и выходным контактом *Y*. В свою очередь, *p*-МОП транзистор P1 включен между напряжением питания V_{DD} и выходным контактом *Y*. Напряжение на входном контакте *A* управляет переходами обоих транзисторов.

Если напряжение на *A* равно 0, то транзистор N1 выключен, а транзистор P1 включен. При этом напряжение на контакте *Y* равно напряжению питания V_{DD}, а не земли, что соответствует логической единице. В этом случае говорят, что *Y* «подтянут» к единице (pulled up). Включенный транзистор P1 хорошо передает логическую единицу (равную напряжению питания), то есть напряжение на контакте *Y* очень близко к V_{DD}. Если же напряжение на контакте *A* равно логической единице, то транзистор N1 включен, а транзистор P1 выключен, и напряжение на контакте *Y* равно напряжению земли, что соответствует логическому нулю. В этом случае говорят, что *Y* «подтянут» к нулю (pulled down). Включенный транзистор N1 хорошо передает логический ноль, то есть напряжение на контакте *Y* очень близко к GND. Сравнение с таблицей истинности на **рис. 1.12** подтверждает, что мы действительно имеем дело с логическим элементом НЕ.

1.7.6. Другие логические элементы на КМОП-транзисторах

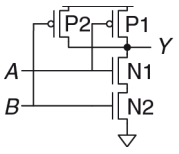


Рис. 1.33 Схема логического элемента И-НЕ с двумя входами

На **рис. 1.33** показана схема для построения с помощью МОП-транзисторов логического элемента И-НЕ с двумя входными контактами. На электронных схемах принято, что если нет никаких дополнительных замечаний или обозначений, то подразумевается, что две линии соединяются друг с другом в том случае, если одна из линий заканчивается в точке пересечения (пересечение в форме буквы Т). Если же обе линии продолжаются за точкой пересечения, то для обозначения контакта этих двух линий в точке пересечения ставится точка. Если точка отсутству-

ет, то это означает, что линии не пересекаются, и одна из линий проходит над другой. На **рис. 1.33** *n*-МОП-транзисторы N1 и N2 соединены последовательно. Причем чтобы замкнуть выходной контакт на землю GND – то есть понизить (pull down) логический уровень, оба этих транзистора должны быть включены. В то время как *p*-МОП-транзисторы P1 и P2 соединены параллельно и только один из них должен быть включен, чтобы соединить выходной контакт с напряжением питания V_{DD} – то есть повысить (pull up) логический уровень. В **табл. 1.6** перечислены все возможные состояния для части схемы, *понижающей логический уровень* (pull-down network), для части схемы, *повышающей логический уровень* (pull-up network), и для выхода. Из **табл. 1.6** видно, что электрическая схема, показанная на **рис. 1.33**, действительно работает как логический элемент И-НЕ. Например, если *A* равно 1 и *B* равно 0, то транзистор N1 включен, но транзистор N2 выключен и блокирует связь контакта *Y* с напряжением земли GND. При этом транзистор P1 выключен, а транзистор P2 включен и соединяет напряжение питания V_{DD} с контактом *Y*. То есть на контакте *Y* мы имеем 1.

Таблица 1.6 Работа логического элемента И-НЕ

A	*B*	Схема понижения логического уровня	Схема повышения логического уровня	*Y*
0	0	Выкл.	Вкл.	1
0	1	Выкл.	Вкл.	1
1	0	Выкл.	Вкл.	1
1	1	Вкл.	Выкл.	0

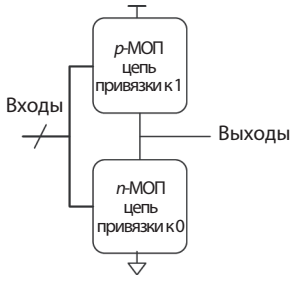


Рис. 1.34 Общая форма инвертирующего логического элемента

Рисунок 1.34 в обобщенном виде показывает блоки, необходимые для построения любого инвертированного логического элемента, такого как НЕ, И-НЕ, ИЛИ-НЕ.

Транзисторы *n*-МОП хорошо передают 0, поэтому схема, понижающая логический уровень, составленная из таких транзисторов, помещается между выходным контактом и землей GND для передачи 0 на выход. Транзисторы *p*-МОП хорошо передают 1, поэтому схема, повышающая логический уровень, составленная из таких транзисторов, помещается между выходным контактом и напряжением питания V_{DD} для передачи 1 на выход. Понижающая и повышающая схемы могут состоять из транзисторов, соединенных как параллельно, так и последовательно. Причем при параллельном соединении транзисторов вся схема включена, если включен хотя бы один из транзисторов. При последовательном соединении схема включена, только если оба транзистора вклю-

чены. Косая черта на входной линии указывает на то, что этот логический элемент имеет несколько входов.

Если и понижающую, и повышающую части схемы включить одновременно, то во всей схеме возникнет короткое замыкание между напряжением питания V_{DD} и землей GND. Сигнал на выходном контакте может оказаться в запретной зоне, а транзисторы, потребляющие при этом большое количество энергии, могут перегореть. С другой стороны, если и понижающую, и повышающую части схемы одновременно выключить, то выходной сигнал будет отключен и от V_{DD}, и от GND. В этом случае говорят, что выходной сигнал *плавает* (floats). Его значение, так же как и в случае одновременно включенных схем, не определено. Обычно наличие плавающего сигнала на выходе системы нежелательно, но в **разделе 2.6** мы рассмотрим, как разработчик может использовать такие сигналы.

Опытные разработчики утверждают, что электронные устройства работают, пока они содержат внутри магический дым. Для подтверждения этой теории они ссылаются на наблюдения, в ходе которых было установлено, что если магический дым по каким-то причинам уходит из устройства наружу, то это устройство прекращает функционировать.

В правильно функционирующем логическом элементе в любой момент времени одна из схем должна быть включена, а другая выключена. При этом напряжение на выходе должно быть или высоким (V_{DD}), или низким (GND). Ни короткое замыкание, ни высокоимпедансное значение сигнала не допускаются. Чтобы гарантировать это условие, пользуются *правилом дополнения проводимости* (conduction complements). Если *n*-МОП-транзисторы в какой-либо цепи соединены последовательно, то *p*-МОП-транзисторы в этой же цепи должны быть соединены параллельно. Если же *n*-МОП-транзисторы соединены параллельно, то *p*-МОП-транзисторы должны быть соединены последовательно.

Пример 1.20 СХЕМА ЛОГИЧЕСКОГО ЭЛЕМЕНТА И-НЕ С ТРЕМЯ ВХОДАМИ

Нарисуйте схему логического элемента И-НЕ с тремя входами, используя КМОП-транзисторы.

Решение Логический элемент И-НЕ должен выдать 0 только в том случае, если все входы равны 1. Следовательно, схема, понижающая логический уровень, должна иметь 3 последовательно включенных *n*-МОП-транзистора. По правилу дополнений (conduction complements) *p*-МОП-транзисторы должны быть включены параллельно. Такой логический элемент показан на **рис. 1.35**. Вы можете удостовериться в правильности функционирования путем проверки таблицы истинности.

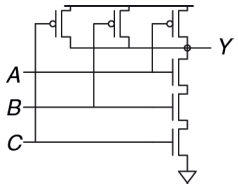


Рис. 1.35 Схема логического элемента И-НЕ с тремя входами

Пример 1.21 СХЕМА ЛОГИЧЕСКОГО ЭЛЕМЕНТА ИЛИ-НЕ С ДВУМЯ ВХОДАМИ

Нарисуйте схему логического элемента ИЛИ-НЕ с двумя входами, используя КМОП-транзисторы.

Решение Логический элемент ИЛИ-НЕ должен выдавать 0, если хотя бы один из входов равен 1. Следовательно, схема, понижающая логиче-

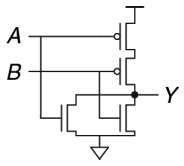


Рис. 1.36 Схема логического элемента ИЛИ-НЕ с двумя входами

ский уровень, должна иметь 2 *n*-МОП-транзистора, включенных параллельно. По правилу дополнений *p*-МОП-транзисторы должны быть включены последовательно. Такой логический элемент показан на **рис. 1.36**.

Пример 1.22 СХЕМА ЛОГИЧЕСКОГО ЭЛЕМЕНТА И С ДВУМЯ ВХОДАМИ

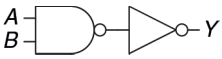


Рис. 1.37 Схема логического элемента И с двумя входами

Нарисуйте схему для логического элемента И с двумя входами, используя КМОП-транзисторы.

Решение Схему И невозможно построить на основе одного КМОП-элемента. При этом построение логических элементов И-НЕ и НЕ – дело довольно простое. Итак, лучший способ построить логический элемент И, применяя КМОП-транзисторы, состоит в том, чтобы использовать И-НЕ, за которым следует НЕ, как показано на **рис. 1.37**.

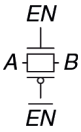


Рис. 1.38 Передаточный логический элемент

1.7.7. Передаточный логический элемент

Иногда разработчику необходим идеальный переключатель, который может одинаково хорошо передавать как 0, так и 1. Вспомним, что *n*-МОП-транзисторы хорошо передают 0, а *p*-МОП-транзисторы хорошо передают 1, и параллельное соединение этих двух транзисторов должно хорошо передавать оба этих значения. На **рис. 1.38** показана такая цепь, называемая *передаточным логическим элементом* (transmission gate), *проходным логическим элементом* (pass gate) или аналоговым ключом. Выводы этого элемента обозначаются A и B, поскольку передача сигнала в таком логическом элементе может идти в двух направлениях, и ни одно из этих направлений не является предпочтительным. Сигналы управления (в англоязычной литературе называемые *enables*) обозначаются EN и $\overline{EN}$. Если EN равен 0, а $\overline{EN}$ равен 1, то оба транзистора выключены. При этом весь передаточный логический элемент выключен, и контакт A не имеет связи с контактом B. Если же EN равен 1, а $\overline{EN}$ равен 0, то передаточный логический элемент включен, и любое логическое значение передается от A к B.

1.7.8. Псевдо-*n*-МОП-логика

Построенный по технологии КМОП логический элемент ИЛИ-НЕ, у которого количество входных контактов равно N, использует N параллельно включенных *n*-МОП-транзисторов и N последовательно включенных *p*-МОП-транзисторов. Последовательно включенные транзисторы передают сигнал медленнее, чем транзисторы, включенные параллельно, аналогично тому, как сопротивление резисторов, включенных последовательно, будет больше, чем сопротивление резисторов, включенных параллельно. Кроме того, *p*-МОП-транзисторы передают сигналы медленнее, чем *n*-МОП-транзисторы, поскольку дырки не могут перемещаться

по кристаллической решетке кремния так же быстро, как электроны. В результате соединенные параллельно *n*-МОП-транзисторы работают быстро, а соединенные последовательно *p*-МОП-транзисторы работают медленно, особенно если их много.

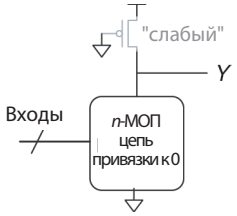


Рис. 1.39 Обобщенный псевдо-*n*-МОП-логический элемент

Как показано на **рис. 1.39**, при использовании *псевдо-n-МОП-логики* (pseudo-nMOS logic), или просто *псевдологики*, медленный стек из *p*-МОП-транзисторов заменяют одним «слабым» *p*-МОП-транзистором, который всегда находится во включенном состоянии. Такой транзистор часто называют *слабым подтягивающим транзистором* (weak pull-up). Физические параметры *p*-МОП-транзистора подбираются таким образом, что этот транзистор до высокого логического уровня (1) выход *Y* «подтягивает слабо» – то есть только в том случае, когда все *n*-МОП-транзисторы выключены. Но если при этом хотя бы один из *n*-МОП-транзисторов включается, то он, превосходя по мощности слабый подтягивающий транзистор, «перетягивает» выход *Y* настолько близко к напряжению земли GND, что на выходе получается логический 0.

Преимущество псевдологики заключается в том, что такую логику можно использовать для создания быстрых логических элементов ИЛИ-НЕ с большим количеством входов. Например, на **рис. 1.40** показан логический элемент ИЛИ-НЕ с четырьмя входами, построенный с использованием псевдологики.

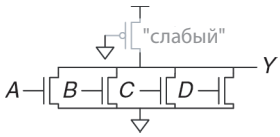


Рис. 1.40 Псевдо-*n*-МОП-логический элемент ИЛИ-НЕ с четырьмя входами

Логические элементы, использующие псевдологику, могут быть очень полезны для построения некоторых видов памяти и логических массивов, описанных в **главе 5**. Недостаток псевдологики – наличие короткого замыкания между питанием V_{DD} и землей GND, когда сигнал на выходе – логический ноль (0). Слабые *p*-МОП- и *n*-МОП-транзисторы выключены. При этом через короткое замыкание постоянно протекает ток, и электрическая энергия от источника питания расходуется впустую. Именно по этой причине псевдо-*n*-МОП-логика используется ограниченно.

Термин «псевдо-*n*-МОП-логика» родился в 70-е годы прошлого века. Тогда существовал производственный процесс для изготовления только *n*-МОП-транзисторов. В то время слабые *n*-МОП-транзисторы использовались для «подтягивания» выходного сигнала до логической единицы (1), поскольку *p*-МОП-транзисторов просто не было.

1.8. Потребляемая мощность

Потребляемая мощность – это количество энергии, потребляемой системой за единицу времени. Энергопотребление имеет большое значение в цифровых системах. Именно потребляемая мощность определяет

время автономной работы без подзарядки батареи любого портативного устройства, такого как сотовый телефон или ноутбук. Не стоит думать, что потребляемая мощность – второстепенный параметр для стационарных устройств. Электричество стоит денег, и к тому же любое устройство может перегреться, если оно потребляет слишком много электроэнергии.

Цифровая система потребляет энергию как в динамическом режиме, когда выполняет какие-либо операции, так и в статическом, когда система находится в состоянии покоя (idle). В динамическом режиме энергия расходуется на зарядку емкостей элементов системы, когда эти элементы переключаются между 0 и 1. И хотя в статическом режиме никаких переключений не происходит, система все равно расходует электрическую энергию.

И сами логические элементы, и проводники, соединяющие эти логические элементы друг с другом, являются конденсаторами и обладают определенной емкостью. Энергия, получаемая от блока питания, которую необходимо затратить на зарядку емкости C до напряжения V_{DD}, равна CV_{DD}^2. Если напряжение на конденсаторе переключается с частотой f (т. е. f раз в секунду), то конденсатор заряжается $f/2$ раза и разряжается $f/2$ раза в секунду. И поскольку в процессе разрядки конденсатор не потребляет энергию от источника питания, то получается, что потребление энергии в динамическом режиме можно рассчитать как

$$P_{dynamic} = {}^1\!/_2 C V_{DD}^2 f. \tag{1.4}$$

Утечка тока в системе происходит, даже если система находится в состоянии покоя. У некоторых типов электронных схем, таких как псевдо-n-МОП-логика, рассмотренных в **разделе 1.7.8**, существует путь, соединяющий напряжение питания V_{DD} с землей GND, через который ток протекает постоянно. Суммарная величина тока, протекающего в системе в ее статическом состоянии I_{DD}, называется *током утечки* (leakage current), или *током покоя* (quiescent supply current). Мощность, потребляемая системой в статическом состоянии, пропорциональна величине тока утечки и может быть рассчитана как

$$P_{static} = I_{DD} V_{DD}. \tag{1.5}$$

Пример 1.23 ПОТРЕБЛЯЕМАЯ МОЩНОСТЬ

Сотовый телефон некоторой модели имеет аккумулятор емкостью 6 Вт-ч и работает от напряжения 1,2 В. Предположим, что во время использования телефон работает на частоте 300 МГц и средняя емкость цифровой схемы телефона в любой конкретный момент составляет 10 нФ (10^{-8} Ф). При работе телефон также выдает сигнал мощностью 3 Вт на антенну. Когда телефон не используется, динамическая потребляемая мощность падает практически до нуля, так как

обработка сигналов отключена. Но телефон также потребляет 40 мА тока покоя независимо от того, работает он или нет. Рассчитайте время, на которое хватит аккумулятора телефона, для случаев:

(а) если телефон не используется;

(б) если телефон используется непрерывно.

Решение Статическая мощность P_{static} равна (0,040 А)(1,2 В) = 48 мВт. Если телефон не используется, это единственное потребление мощности, поэтому время жизни аккумулятора равно (6 Вт-ч)/(0,048 Вт) = 125 ч (примерно 5 дней). В случае если телефон используется, динамическая мощность $P_{dynamic}$ равна $(0,5)(10^{-8}\ \text{Ф})(1,2\ \text{В})^2(3\times10^8\ \text{Гц}) = 2,16$ Вт. Общая мощность, являющаяся суммой $P_{dynamic}$, P_{static} и мощности вещания, составит 2,16 Вт + 0,048 Вт + 3 Вт = 5,2 Вт, поэтому время жизни аккумулятора будет равно 6 Вт-ч/5,2 Вт = 1,15 ч. В этом примере реальная работа телефона представлена в несколько упрощенном виде, но тем не менее он иллюстрирует ключевые идеи, связанные с мощностью потребления.

1.9. Краткий обзор главы 1 и того, что нас ждет впереди

В этом мире существует 10 видов людей: те, кто знакомы с двоичной системой счисления, и те, кто не знают о ней ничего.

В этой главе мы описали основные концепции, необходимые для понимания и разработки сложных электронных систем. И хотя физические величины в реальном мире в большинстве своем аналоговые – то есть изменяются непрерывно, разработчики цифровых систем ограничиваются рассмотрением конечного подмножества дискретных величин непрерывно меняющихся сигналов. В частности, логические переменные могут принимать только два значения – 0 и 1, которые еще называются ЛОЖЬ (FALSE) и ИСТИНА (TRUE), или НИЗКИЙ уровень логического сигнала (LOW) и ВЫСОКИЙ уровень логического сигнала (HIGH). Логические элементы определенным образом преобразуют сигналы с одного или нескольких двоичных входов в двоичный сигнал на выходе. Некоторые из наиболее часто используемых логических элементов перечислены ниже:

- **НЕ:** имеет на выходе значение ИСТИНА, если сигнал на входе имеет значение ЛОЖЬ.
- **И:** имеет на выходе значение ИСТИНА, если все сигналы на входе имеют значение ИСТИНА.
- **ИЛИ:** имеет на выходе значение ИСТИНА, если хотя бы один сигнал на входе имеет значение ИСТИНА.
- **Исключающее ИЛИ:** имеет на выходе значение ИСТИНА, если нечетное количество сигналов на входе имеет значение ИСТИНА.

Для построения логических элементов обычно используются транзисторы КМОП, которые, по сути, являются переключателями с электрическим управлением. Транзистор *n*-МОП включается, если затвор находится под напряжением V_{DD}, что соответствует логической единице. Транзистор *p*-МОП включается, если затвор находится под напряжением GND, что соответствует логическому нулю.

В **главах 2–5** мы продолжим изучение цифровой логики. В **главе 2** рассматривается *комбинационная логика* (combinational logic), в которой предполагается, что сигнал на выходе логического элемента зависит только от состояний на входах этого элемента в конкретный момент времени. Те логические элементы, которые мы уже рассмотрели в этой книге, могут служить в качестве примера использования комбинационной логики. Из **главы 2** вы также поймете, как можно разработать схему из нескольких логических элементов таким образом, чтобы все возможные состояния этой схемы соответствовали состояниям, заранее описанным в таблице истинности или с помощью логического уравнения.

Глава 3 описывает *последовательностную логику* (sequential logic). Такая логика уже допускает, что результат на выходе логического элемента зависит как от текущего состояния на входе, так и от прошлых его состояний. *Регистр* (register) – это наиболее распространенный элемент последовательностной логики, который «запоминает» предыдущее состояние на своем входе. *Конечный автомат* (finite state machines), построенный на базе регистров и комбинационной логики, является мощным средством для создания сложных систем на системной основе. В **главе 3** мы также рассмотрим временные соотношения сигналов в цифровой системе, чтобы определить максимально возможную скорость, на которой эта система может нормально работать.

Глава 4 рассматривает *языки описания аппаратуры* (hardware description languages, HDL). Языки HDL – родственники обычных языков программирования, но используются они, по большей части, для моделирования и создания аппаратного, а не программного обеспечения. Большинство современных цифровых систем были разработаны с использованием HDL. SystemVerilog и VHDL – два наиболее распространенных языка для описания и верификации аппаратуры, и оба они рассматриваются в этой книге. *VHDL* (Very high-speed integrated circuits Hardware Description Language) переводится как *язык для описания и верификации аппаратуры на очень высокоскоростных интегральных схемах*.

Глава 5 описывает другие элементы комбинационной и последовательностной логик, такие как *сумматоры* (adders), *умножители* (multipliers) и *блоки памяти* (memories).

Глава 6 посвящена описанию компьютерной архитектуры. Она описывает процессор RISC-V – недавно разработанный микропроцессор

с открытым исходным кодом, который становится все более популярным в промышленности и научных кругах. Архитектура RISC-V определяется его регистрами и набором инструкций на языке ассемблера. Вы узнаете, как разрабатывать программы для процессора RISC-V на языке ассемблера, то есть общаться с этим процессором на его родном языке.

Главы 7 и **8** перекидывают мостик от цифровой логики к компьютерной архитектуре. **Глава 7** исследует микроархитектуру – то есть организацию отдельных строительных блоков, таких как сумматоры и регистры, необходимых для построения работающего процессора. Эта глава научит вас навыкам, необходимым для разработки вашего собственного процессора RISC-V. Более того, в **главе 7** мы рассмотрим три микроархитектуры, иллюстрирующие различные компромиссы между производительностью процессора и затратами на его производство. Долгое время производительность процессоров росла по экспоненте, требуя все более изощренных блоков памяти, чтобы удовлетворить постоянно растущий спрос на данные. **Глава 8** погрузит вас в особенности архитектуры блоков памяти, а также позволит понять, как компьютеры связываются с периферийными устройствами, такими как клавиатура или принтер.

Упражнения

Упражнение 1.1 Объясните не менее трех уровней абстракции, которые используются:

a) биологами, изучающими работу клеток;
b) химиками, изучающими состав какого-либо материала.

Ваше объяснение не должно быть длиннее одного абзаца.

Упражнение 1.2 Объясните, как методы иерархичности, модульности и регулярности могут быть использованы:

a) конструкторами автомобилей;
b) каким-либо бизнесом для управления ежедневными операциями.

Ваше объяснение не должно быть длиннее одного абзаца.

Упражнение 1.3 Бен Битдидл[1] строит дом. Объясните ему, как он может использовать принципы иерархичности, модульности и регулярности, чтобы сэкономить время и ресурсы.

Упражнение 1.4 Допустим, что напряжение аналогового сигнала в нашей системе меняется в пределах от 0 В до 5 В. Если мы можем измерить это напряже-

[1] Бен Битдидл (Ben Bitdiddle) – персонаж, созданный Стивом Уордом (Steve Ward) в 1970-х годах и с той поры широко используемый в качестве героя сборников задач в Массачусетском технологическом институте (Massachusetts Institute of Technology, MIT) и вне его. Фамилия Бена происходит от термина «bit diddling», который можно перевести как «битовое жонглирование» – программирование на уровне машинных кодов с манипулированием битами, флагами, полубайтами и другими элементами размером меньше слова. – *Прим. перев.*

ние с точностью до ±50 милливольт, какое максимальное количество информации в битах этот сигнал может передавать?

Упражнение 1.5 На стене висят старые часы с отломанной минутной стрелкой.

a) Если, используя только часовую стрелку, вы можете определить текущее время с точностью до 15 минут, то сколько битов информации о времени вы можете получить, глядя на эти часы?
b) Если вы будете знать, какая сейчас половина дня – до или после полудня, то сколько дополнительных битов информации о текущем времени вы получите?

Упражнение 1.6 Примерно 4000 лет назад вавилоняне разработали шестидесятеричную (по основанию 60) систему счисления. Сколько битов информации передает одна шестидесятеричная цифра? Как можно записать число 4000_{10}, используя шестидесятеричную систему счисления?

Упражнение 1.7 Как много различных чисел может быть представлено 16 битами?

Упражнение 1.8 Какое максимальное значение может быть представлено 32-разрядным двоичным числом?

Упражнение 1.9 Какое максимальное 16-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже:

a) двоичное число без знака (unsigned);
b) дополнительный код (two's complement);
c) прямой код (sign / magnitude).

Упражнение 1.10 Какое максимальное 32-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже:

a) двоичное число без знака (unsigned);
b) дополнительный код (two's complement);
c) прямой код (sign / magnitude).

Упражнение 1.11 Какое минимальное (наименьшее отрицательное) 16-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже:

a) двоичное число без знака (unsigned);
b) дополнительный код (two's complement);
c) прямой код (sign / magnitude).

Упражнение 1.12 Какое минимальное (наименьшее отрицательное) 32-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже:

a) двоичное число без знака (unsigned);
b) дополнительный код (two's complement);
c) прямой код (sign / magnitude).

Упражнение 1.13 Преобразуйте следующие двоичные числа без знака в десятичные.

a) 1010_2
b) 110110_2
c) 11110000_2
d) 0001000101001112

Упражнение 1.14 Преобразуйте следующие двоичные числа без знака в десятичные.

a) 1110_2
b) 100100_2
c) 11010111_2
d) 0111010101001002

Упражнение 1.15 Преобразуйте двоичные числа без знака из **упражнения 1.13** в шестнадцатеричные.

Упражнение 1.16 Преобразуйте двоичные числа без знака из **упражнения 1.14** в шестнадцатеричные.

Упражнение 1.17 Преобразуйте следующие шестнадцатеричные числа в десятичные.

a) $A5_{16}$
b) $3B_{16}$
c) $FFFF_{16}$
d) $D0000000_{16}$

Упражнение 1.18 Преобразуйте следующие шестнадцатеричные числа в десятичные.

a) $4E_{16}$
b) $7C_{16}$
c) $ED3A_{16}$
d) $403FB001_{16}$

Упражнение 1.19 Преобразуйте шестнадцатеричные числа из **упражнения 1.17** в двоичные числа без знака.

Упражнение 1.20 Преобразуйте шестнадцатеричные числа из **упражнения 1.18** в двоичные числа без знака.

Упражнение 1.21 Преобразуйте следующие двоичные числа, представленные в дополнительном коде, в десятичные.

a) 1010_2
b) 110110_2
c) 01110000_2
d) 10011111_2

Упражнение 1.22 Преобразуйте следующие двоичные числа, представленные в дополнительном коде, в десятичные.

a) 1110_2
b) 100011_2
c) 01001110_2
d) 10110101_2

Упражнение 1.23 Преобразуйте двоичные числа из **упражнения 1.21** в десятичные, считая, что эти двоичные числа представлены не в дополнительном, а в прямом коде.

Упражнение 1.24 Преобразуйте двоичные числа из **упражнения 1.22** в десятичные, считая, что эти двоичные числа представлены не в дополнительном, а в прямом коде.

Упражнение 1.25 Преобразуйте следующие десятичные числа в двоичные числа без знака.

a) 42_{10}
b) 63_{10}
c) 229_{10}
d) 845_{10}

Упражнение 1.26 Преобразуйте следующие десятичные числа в двоичные числа без знака.

a) 14_{10}
b) 52_{10}
c) 339_{10}
d) 711_{10}

Упражнение 1.27 Преобразуйте десятичные числа из **упражнения 1.25** в шестнадцатеричные.

Упражнение 1.28 Преобразуйте десятичные числа из **упражнения 1.26** в шестнадцатеричные.

Упражнение 1.29 Преобразуйте следующие десятичные числа в 8-битные двоичные числа, представленные в дополнительном коде. Укажите, произошло ли переполнение.

a) 42_{10}
b) -63_{10}
c) 124_{10}
d) -128_{10}
e) 133_{10}

Упражнение 1.30 Преобразуйте следующие десятичные числа в 8-битные двоичные числа, представленные в дополнительном коде. Укажите, произошло ли переполнение.

a) 24_{10}
b) -59_{10}
c) 128_{10}
d) -150_{10}
e) 127_{10}

Упражнение 1.31 Преобразуйте десятичные числа из **упражнения 1.29** в 8-битные двоичные числа, представленные в прямом коде.

Упражнение 1.32 Преобразуйте десятичные числа из **упражнения 1.30** в 8-битные двоичные числа, представленные в прямом коде.

Упражнение 1.33 Преобразуйте следующие 4-разрядные двоичные числа, представленные в дополнительном коде, в 8-разрядные двоичные числа, представленные в дополнительном коде:

a) 0101_2
b) 1010_2

Упражнение 1.34 Преобразуйте следующие 4-разрядные двоичные числа, представленные в дополнительном коде, в 8-разрядные двоичные числа, представленные в дополнительном коде:

a) 0111_2
b) 1001_2

Упражнение 1.35 Преобразуйте 4-разрядные двоичные числа из **упражнения 1.33** в 8-разрядные, считая, что это двоичные числа без знака.

Упражнение 1.36 Преобразуйте 4-разрядные двоичные числа из **упражнения 1.34** в 8-разрядные, считая, что это двоичные числа без знака.

Упражнение 1.37 Система счисления по основанию 8 называется *восьмеричной* (*octal*). Представьте каждое из чисел в **упражнении 1.25** в восьмеричном виде.

Упражнение 1.38 Система счисления по основанию 8 называется восьмеричной. Представьте каждое из чисел в **упражнении 1.26** в восьмеричном виде.

Упражнение 1.39 Преобразуйте каждое из следующих восьмеричных чисел в двоичное, шестнадцатеричное и десятичное:

a) 42_8
b) 63_8
c) 255_8
d) 3047_8

Упражнение 1.40 Преобразуйте каждое из следующих восьмеричных чисел в двоичное, шестнадцатеричное и десятичное:

a) 23_8
b) 45_8
c) 371_8
d) 2560_8

Упражнение 1.41 Сколько 5-разрядных двоичных чисел, представленных в дополнительном коде, имеют значение большее, чем 0? Сколько – меньшее, чем 0? Каким будет правильный ответ в случае 5-разрядных двоичных чисел, представленных в прямом коде?

Упражнение 1.42 Сколько 7-разрядных двоичных чисел, представленных в дополнительном коде, имеют значение большее, чем 0? Сколько меньшее, чем 0? Каким будет правильный ответ в случае 7-разрядных двоичных чисел, представленных в прямом коде?

Упражнение 1.43 Сколько байтов в 32-битном слове? Сколько полубайтов?

Упражнение 1.44 Сколько байтов в 64-битном слове?

Упражнение 1.45 Если DSL-модем работает со скоростью 768 кбит/с, сколько байтов он может передать за 1 минуту?

Упражнение 1.46 USB3.0 передает данные со скоростью 5 Гбит/с. Сколько байтов он может передать за 1 минуту?

Упражнение 1.47 Производители жестких дисков измеряют объемы данных в мегабайтах, что означает 10^6 байт, и гигабайтах, что означает 10^9 байт. Сколько гигабайтов музыки вы можете сохранить на 50-гигабайтном жестком диске?

Упражнение 1.48 Без использования калькулятора рассчитайте приблизительное значение 2^{31}.

Упражнение 1.49 Память процессора Pentium II организована как прямоугольный массив битов, состоящий из 2^8 строк и 2^9 колонок. Без использования калькулятора рассчитайте приблизительное количество битов в этом массиве.

Упражнение 1.50 Нарисуйте цифровую шкалу, аналогичную изображенной на **рис. 1.11**, для 3-битного двоичного числа, представленного в дополнительном и прямом кодах.

Упражнение 1.51 Нарисуйте цифровую шкалу, аналогичную изображенной на **рис. 1.11**, для 2-битного двоичного числа, представленного в дополнительном и прямом кодах.

Упражнение 1.52 Сложите следующие двоичные числа без знака:

a) $1001_2 + 0100_2$
b) $1101_2 + 1011_2$

Укажите, произошло ли переполнение 4-битного регистра.

Упражнение 1.53. Сложите следующие двоичные числа без знака:

a) $10011001_2 + 01000100_2$
b) $11010010_2 + 10110110_2$

Укажите, произошло ли переполнение 8-битного регистра.

Упражнение 1.54 Выполните **упражнение 1.52**, считая, что двоичные числа в этом упражнении представлены в дополнительном коде.

Упражнение 1.55 Выполните **упражнение 1.53**, считая, что двоичные числа в этом упражнении представлены в дополнительном коде.

Упражнение 1.56 Преобразуйте следующие десятичные числа в 6-битные двоичные числа, представленные в дополнительном коде, и сложите их:

a) $16_{10} + 9_{10}$
b) $27_{10} + 31_{10}$
c) $-4_{10} + 19_{10}$
d) $3_{10} + -32_{10}$
e) $-16_{10} + -9_{10}$
f) $-27_{10} + -31_{10}$

Укажите, произошло ли переполнение 6-битного регистра.

Упражнение 1.57 Преобразуйте следующие десятичные числа в 6-битные двоичные числа, представленные в дополнительном коде, и сложите их:

a) $7_{10} + 13_{10}$
b) $17_{10} + 25_{10}$
c) $-26_{10} + 8_{10}$
d) $31_{10} + -14_{10}$
e) $-19_{10} + -22_{10}$
f) $-2_{10} + -29_{10}$

Укажите, произошло ли переполнение 6-битного регистра.

Упражнение 1.58 Сложите следующие шестнадцатеричные числа без знака:

a) $7_{16} + 9_{16}$
b) $13_{16} + 28_{16}$
a) $AB_{16} + 3E_{16}$
b) $8F_{16} + AD_{16}$

Укажите, произошло ли переполнение 8-битного регистра.

Упражнение 1.59 Сложите следующие шестнадцатеричные числа без знака:

a) $22_{16} + 8_{16}$
b) $73_{16} + 2C_{16}$
c) $7F_{16} + 7F_{16}$
d) $C2_{16} + A4_{16}$

Укажите, произошло ли переполнение 8-битного регистра.

Упражнение 1.60 Преобразуйте следующие десятичные числа в 5-разрядные двоичные числа, представленные в дополнительном коде, и вычтите одно число из другого:

a) $9_{10} - 7_{10}$
b) $12_{10} - 15_{10}$
c) $-6_{10} - 11_{10}$
d) $4_{10} - -8_{10}$

Укажите, произошло ли переполнение 5-битного регистра.

Упражнение 1.61 Преобразуйте следующие десятичные числа в 6-разрядные двоичные числа, представленные в дополнительном коде, и вычтите одно число из другого:

a) $18_{10} - 12_{10}$
b) $30_{10} - 9_{10}$
c) $-28_{10} - 3_{10}$
d) $-16_{10} - 21_{10}$

Укажите, произошло ли переполнение 6-битного регистра.

Упражнение 1.62 В *N-битной двоичной системе счисления со смещением B* (*N*-bit binary number system with *bias B*) положительные и отрицательные числа представляются как значения этих чисел в обычной двоичной системе плюс смещение *B*. Например, для 5-битной двоичной системы счисления со смещением 15 число 0 представляется как 01111, а число 1 представляется как 10000 и так

далее. Системы счисления со смещением иногда используются для выполнения математических операций с плавающей запятой, которые будут рассмотрены в **главе 5**. Ответьте на следующие вопросы применительно к 8-битной системе счисления со смещением 127_{10}:

a) Какое десятичное значение соответствует двоичному числу 10000010_2?
b) Какое двоичное число соответствует значению 0?
c) Как в такой системе будет выглядеть минимальное отрицательное двоичное число, и каким будет его десятичный эквивалент?
d) Как в такой системе будет выглядеть максимальное положительное двоичное число, и каким будет его десятичный эквивалент?

Упражнение 1.63 Нарисуйте цифровую шкалу, аналогичную изображенной на **рис. 1.11**, для 3-битного двоичного числа со смещением, равным 3. Что такое система счисления со смещением, объясняется в **упражнении 1.62**.

Упражнение 1.64 В двоично-десятичной системе счисления (binary-coded decimal system, BCD) 4 бита используются для представления десятичных чисел от 0 до 9. Например, 37_{10} записывается как 00110111_{BCD}.
Ответьте на следующие вопросы применительно к двоично-десятичной системе счисления.

a) Как будет выглядеть 289_{10} в двоично-десятичной системе счисления?
b) Как выглядит десятичный эквивалент 100101010001_{BCD}?
c) Как выглядит двоичный эквивалент 01101001_{BCD}?
d) Какие, по-вашему мнению, преимущества имеет двоично-десятичная система счисления?

Упражнение 1.65 Ответьте на следующие вопросы применительно к двоично-десятичной системе счисления.

a) Как будет выглядеть 371_{10} в двоично-десятичной системе счисления?
b) Как выглядит десятичный эквивалент 000110000111_{BCD}?
c) Как выглядит двоичный эквивалент 10010101_{BCD}?
d) Какие, на ваш взгляд, недостатки имеет двоично-десятичная система счисления по сравнению с двоичной?

Что такое двоично-десятичная система счисления со смещением, объясняется в **упражнении 1.64**.

Упражнение 1.66 Марсианская летающая тарелка потерпела крушение на кукурузном поле в штате Небраска. Следователи ФБР обнаружили среди обломков руководство по космической навигации с формулами, записанными в марсианской системе счисления. Одна из формул выглядит следующим образом: 325 + 42 = 411. Если эта формула записана без ошибок, сколько пальцев на руке марсианина вы бы ожидали увидеть?

Упражнение 1.67 У Бена Битдидла и Алисы П. Хакер[1] возник спор. Бен утверждает, что у всех целых чисел, которые больше нуля и кратны шести, есть точно две единицы в двоичном представлении. Алиса не согласна. По ее мнению,

[1] В англоязычном варианте имя Alyssa P. Hacker созвучно выражению «a LISP hacker», т. е. LISP-хакер (LISP – семейство функциональных языков программирования). – *Прим. перев.*

все такие числа имеют четное количество единиц в их представлении. Вы согласны с Беном, с Алисой, с обоими или ни с кем из них? Объясните.

Упражнение 1.68 Бен Битдидл и Алиса П. Хакер снова спорят. Бен говорит: «Я могу получить представление числа в дополнительном коде путем вычитания 1, а затем инвертируя все биты результата». Алиса отвечает: «Нет, я могу это сделать путем проверки каждого бита, начиная с наименее значимых. Когда встречу первую 1, инвертирую каждый последующий бит». Вы согласны с Беном, или с Алисой, или с обоими, или ни с кем? Объясните.

Упражнение 1.69 Напишите программу на вашем любимом языке (например, C, Java, Python) для преобразования двоичных чисел в десятичные. Пользователь должен ввести беззнаковое двоичное число. Программа должна распечатать его десятичный эквивалент.

Упражнение 1.70 Повторите **упражнение 1.69**, но для преобразования чисел в системе счисления с произвольной базой b1 в числа в системе счисления с другой базой b2. Обеспечьте поддержку баз до 16, для цифр больше 9 используйте буквы алфавита. Пользователь должен ввести b1, b2, а затем число в системе счисления с базой b1. Программа должна напечатать эквивалентное число в системе счисления с базой b2.

Упражнение 1.71 Нарисуйте обозначение, логическую функцию и таблицу истинности для:

a) логического элемента ИЛИ с тремя входами;
b) логического элемента Исключающее ИЛИ с тремя входами;
c) логического элемента Исключающее ИЛИ-НЕ с четырьмя входами.

Упражнение 1.72 Нарисуйте обозначение, логическую функцию и таблицу истинности для:

a) логического элемента ИЛИ с четырьмя входами;
b) логического элемента Исключающее ИЛИ-НЕ с тремя входами;
c) логического элемента И-НЕ с пятью входами.

Упражнение 1.73 *Мажоритарный логический элемент* выдает значение ИСТИНА тогда и только тогда, когда более половины его входов имеют значение ИСТИНА. Заполните таблицу истинности для мажоритарного логического элемента, показанного на **рис. 1.41**.

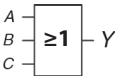


Рис. 1.41 Мажоритарный логический элемент с тремя входами

Упражнение 1.74 Логический элемент И-ИЛИ (AND-OR, AO) с тремя входами, показанный на **рис. 1.42**, выдает значение ИСТИНА, если входы *A* и *B* имеют значение ИСТИНА или вход *C* имеет значение ИСТИНА. Заполните таблицу истинности для этого логического элемента.

Рис. 1.42 Логический элемент И-ИЛИ с тремя входами

Упражнение 1.75 Логический элемент Инвертированный ИЛИ-И (OR-AND-INVERTOR, OAI) с тремя входами, показанный на **рис. 1.43**, выдает значение ЛОЖЬ, если вход *C* имеет значение ИСТИНА и входы *A* или *B* имеют значение ИСТИНА. Иначе логический элемент выдает значение ИСТИНА. Заполните таблицу истинности для этого логического элемента.

Рис. 1.43 Инвертированный логический элемент И-ИЛИ с тремя входами

Упражнение 1.76 Имеется 16 разных таблиц истинности для логических функций от двух переменных. Исследуйте эти таблицы, давая каждой одно короткое описательное имя (например, ИЛИ, И-НЕ и т. д.).

Упражнение 1.77 Сколько существует различных таблиц истинности для логических функций от N переменных?

Упражнение 1.78 Можно ли назначить логические уровни так, чтобы устройство с передаточными характеристиками, показанными на **рис. 1.44**, могло служить в качестве инвертора? Если да, то какими являются входные и выходные низкие и высокие уровни (V_{IL}, V_{OL}, V_{IH} и V_{OH}) и уровни шума (N_{ML} и N_{MH})? Если это не так, то объясните, почему.

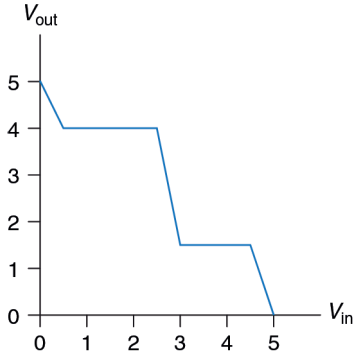


Рис. 1.44 Передаточные характеристики

Упражнение 1.79 Повторите **упражнение 1.78** для передаточных характеристик, показанных на **рис. 1.45**.

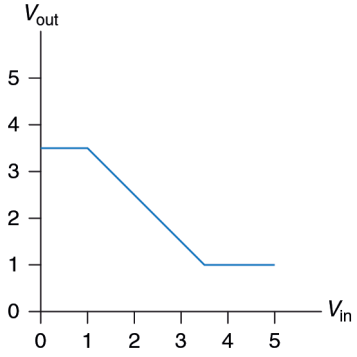


Рис. 1.45 Передаточные характеристики

Упражнение 1.80 Можно ли назначить логические уровни так, чтобы устройство с передаточными характеристиками, показанными на **рис. 1.46**, могло служить в качестве буфера? Если да, то какими являются входные и выходные низкие и высокие уровни (V_{IL}, V_{OL}, V_{IH}, и V_{OH}) и уровни шума (N_{ML} и N_{MH})? Если это не так, то объясните, почему.

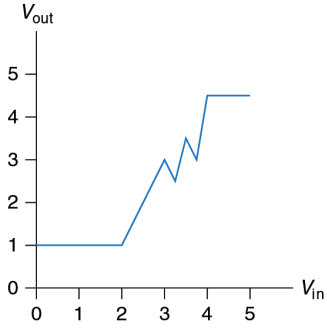


Рис. 1.46 Передаточные характеристики

Упражнение 1.81 Бен Битдидл придумал схему с передаточными характеристиками, показанными на **рис. 1.47**, чтобы использовать ее в качестве буфера. Будет ли эта схема работать? Почему да или почему нет? Он утверждает, что она совместима с низковольтными КМОП- и НТТЛ-структурами. Может ли буфер Бена корректно получать входные сигналы от этих логических элементов? Может ли его выход управлять этими логическими элементами? Объясните.

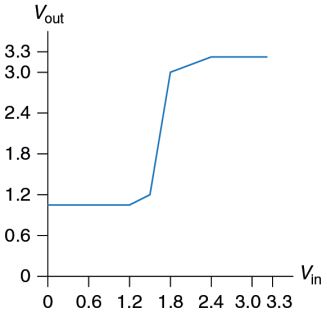


Рис. 1.47 Передаточные характеристики буфера Бена

Упражнение 1.82 Во сне Бен Битдидл увидел логический элемент с двумя входами и передаточной функцией, показанной на **рис. 1.48**. Входы обозначены как A и B, а выходной сигнал – Y.

a) Какого типа логический элемент он увидел?
b) Каковы приблизительные значения высокого и низкого логических уровней?

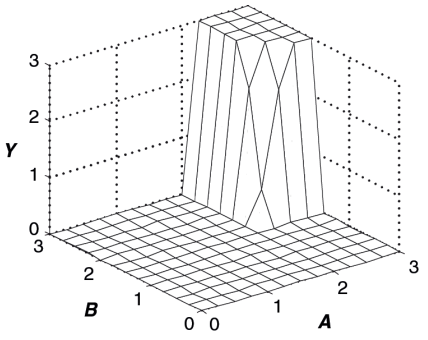


Рис. 1.48 Передаточные характеристики с двумя входами

Упражнение 1.83 Повторите **упражнение 1.82** для **рис. 1.49**.

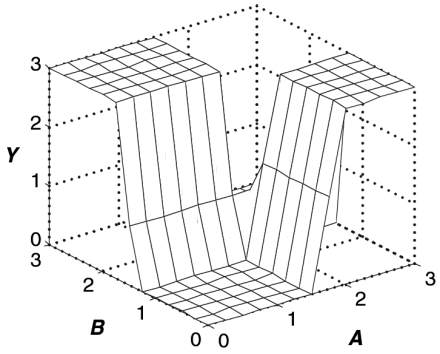


Рис. 1.49 Передаточные характеристики с двумя входами

Упражнение 1.84 Сделайте набросок схемы на уровне транзисторов для следующих КМОП-логических элементов. Используйте минимальное количество транзисторов.

a) Логический элемент И-НЕ с четырьмя входами.
b) Логический элемент Инвертированный ИЛИ-И с тремя входами (**упражнение 1.75**).
c) Логический элемент И-ИЛИ с тремя входами (**упражнение 1.74**).

Упражнение 1.85 Сделайте эскиз схемы на уровне транзисторов для следующих КМОП-логических элементов. Используйте минимальное количество транзисторов.

a) Логический элемент ИЛИ-НЕ с тремя входами.
b) Логический элемент И с тремя входами.
c) Логический элемент ИЛИ с двумя входами.

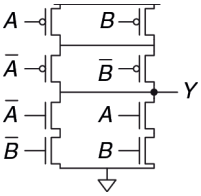


Рис. 1.50 Таинственная схема

Упражнение 1.86 Миноритарный логический элемент выдает значение ИСТИНА тогда и только тогда, когда меньше половины его входов имеют значение ИСТИНА. В противном случае он выдает значение ЛОЖЬ. Сделайте эскиз схемы на уровне транзисторов для КМОП-миноритарного логического элемента. Используйте минимальное количество транзисторов.

Упражнение 1.87 Напишите таблицу истинности для функции логического элемента на **рис. 1.50**. Таблица должна иметь два входа *A* и *B*. Как называется эта функция?

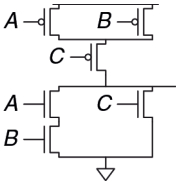


Рис. 1.51 Таинственная схема

Упражнение 1.88 Напишите таблицу истинности для функции логического элемента на **рис. 1.51**. Таблица должна иметь три входа *A*, *B* и *C*.

Упражнение 1.89 Реализуйте следующие логические элементы с тремя входами, используя только псевдо-*n*-МОП-логические элементы. Используйте минимальное количество транзисторов:

a) логический элемент ИЛИ-НЕ;
b) логический элемент И-НЕ;
c) логический элемент И.

Упражнение 1.90 Резисторно-транзисторная логика (*РТЛ*) использует *n*-МОП-транзисторы для выдачи значения НИЗКИЙ (LOW) и резистор с малым сопротивлением для выдачи значения ВЫСОКИЙ (HIGH), когда ни один из путей к заземлению не активен. Логический элемент НЕ, построенный с помощью РТЛ, показан на **рис. 1.52**. Сделайте эскиз схемы РТЛ-логического элемента ИЛИ-НЕ с тремя входами. Используйте минимальное количество транзисторов.

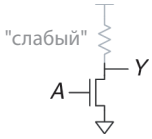


Рис. 1.52 Логический элемент НЕ

Вопросы для собеседования

Эти вопросы часто задают разработчикам цифровых систем в ходе собеседования при устройстве на работу.

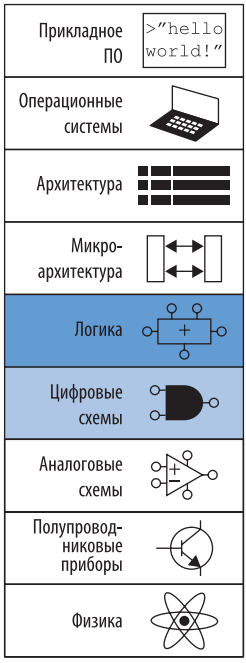
Вопрос 1.1 Сделайте эскиз КМОП-схемы на уровне транзисторов для логического элемента ИЛИ-НЕ с четырьмя входами.

Вопрос 1.2 Король получил 64 золотые монеты в виде налогов, но у него есть основания полагать, что одна из них является поддельной. Король поручил вам выявить поддельную монету. У вас есть весы, на чашки которых можно положить сколько угодно монет на каждой стороне. Сколько раз вам нужно произвести взвешивание, чтобы найти более легкую фальшивую монету?

Вопрос 1.3 Профессор, преподаватель, студент, занимающийся разработкой цифровых схем, и первокурсник-чемпион по бегу хотят перейти шаткий мост темной ночью. Мост настолько плохой, что безопасно по нему могут одновременно пройти только два человека. У нашей группы есть всего один фонарик, без него идти страшно, а мост слишком длинный, чтобы перебросить через него фонарик, так что после каждого перехода кто-то должен его перенести обратно к оставшимся людям. Первокурсник может пересечь мост за 1 минуту. Старший студент может пересечь мост за 2 минуты. Преподаватель может пересечь мост за 5 минут. Профессор всегда отвлекается, поэтому ему нужно 10 минут, чтобы пересечь мост. Как организовать переход, чтобы все перешли через мост за кратчайшее время?

Глава 2

Разработка комбинационной логики



2.1. Введение

В цифровой электронике под *схемой* понимают электрическую цепь, обрабатывающую дискретные сигналы. Такую схему можно рассматривать как «черный ящик», как показано на **рис. 2.1**, при этом схема имеет:

- один или более дискретных *входов*;
- один или более дискретных *выходов*;
- *функциональную спецификацию* (functional specification), описывающую взаимосвязь между входами и выходами;

- *временну́ю спецификацию* (timing specification), описывающую задержку между изменением сигналов на входе и откликом выходного сигнала.

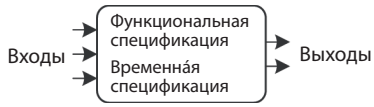


Рис. 2.1 Схема как «черный ящик» с входами, выходами и спецификациями

Если заглянуть внутрь такого «черного ящика», мы увидим, что схемы состоят из соединений, также называемых *узлами* (nodes), и элементов. Элемент представляет собой схему с входами, выходами и спецификацией. Соединение – это проводник, напряжение на котором соответствует дискретной переменной. Соединения подразделяются на входы, выходы и внутренние соединения. Входы получают сигналы извне. Выходы отправляют сигналы во внешний мир. Соединения, которые не являются входами или выходами, называются внутренними соединениями. На **рис. 2.2** показана электронная схема с тремя элементами E1, E2 и E3 и шестью соединениями. Соединения *A*, *B* и *C* – входы, *Y* и *Z* – выходы, а n1 – внутреннее соединение между E1 и E3.

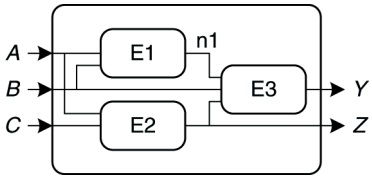


Рис. 2.2 Элементы и соединения

Цифровые схемы разделяются на *комбинационные* (combinational) и *последовательностные* (sequential). Выходы комбинационных схем зависят только от текущих значений на входах; другими словами, такие схемы комбинируют текущие значения входных сигналов для вычисления значения на выходе. Например, логический элемент – это комбинационная схема. Выходы последовательностных схем зависят и от текущих, и от предыдущих значений на входах, то есть зависят от последовательности изменения входных сигналов. У комбинационных схем, в отличие от последовательностных схем, память отсутствует. Данная глава посвящена комбинационным схемам, а в **главе 3** мы рассмотрим последовательностные схемы.

Функциональная спецификация комбинационной схемы описывает зависимость значений на выходах от текущих входных значений. Временная спецификация комбинационной схемы состоит из нижней и верхней граничных значений задержки сигнала на пути от входа к выходу. В этой главе мы сначала рассмотрим функциональную спецификацию, а потом вернемся к временной.

На **рис. 2.3** показана комбинационная схема с двумя входами и одним выходом. Входы A и B расположены слева, справа изображен выход Y. Символ ℄ в прямоугольнике означает, что этот элемент реализован с использованием исключительно комбинационной логики. В этом примере функция F определена как «ИЛИ»: $Y = F(A, B) = A + B$.

Другими словами, мы говорим, что выход Y – это функция двух входов A и B, а именно $Y = A$ ИЛИ B. На **рис. 2.4** показаны два возможных способа построения комбинационной логической схемы, приведенной на **рис. 2.3**. Как будет неоднократно показано в этой книге, зачастую существует множество способов реализации одной и той же функции. Вы сами выбираете, как реализовать требуемую функцию, исходя из имеющихся в распоряжении «строительных блоков», а также ваших проектных ограничений. Эти ограничения часто включают в себя занимаемую на чипе площадь, скорость работы, потребляемую мощность и время разработки.

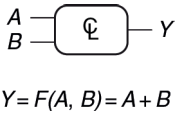


Рис. 2.3 Комбинационная логическая схема

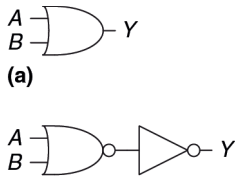


Рис. 2.4 Два варианта схемы ИЛИ

На **рис. 2.5** показана комбинационная схема с несколькими выходами. Данная комбинационная схема называется полным сумматором, мы еще вернемся к ней в **разделе 5.2.1**. Два уравнения определяют значения на выходах S и C_{out} как функции входных сигналов A, B и C_{in}.

Для упрощения чертежей мы часто используем перечеркнутую косой чертой линию и число рядом с ней для обозначения *шины* (bus), то есть группы сигналов. Число показывает, сколько сигналов в шине[1]. Например, на **рис. 2.6 (а)** показан блок комбинационной логики с тремя входами и двумя выходами. Если количество разрядов не имеет значения или очевидно из контекста, то косая черта может быть без значения количества рядом.

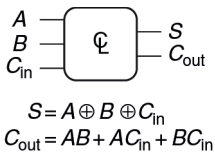


Рис. 2.5 Комбинационная схема с множеством выходов

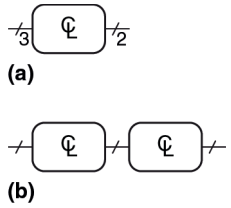


Рис. 2.6 Обозначение шин на схемах

[1] Это число обычно называется *шириной шины*. – *Прим. перев.*

Правила комбинационной композиции схем являются достаточными, но не строго необходимыми. Некоторые схемы, не подчиняющиеся этим правилам, все же являются комбинационными, поскольку значения их выходов зависят только от текущих значений на входах. Бывает довольно сложно определить, являются ли некоторые нетипичные схемы комбинационными или нет, поэтому обычно при разработке комбинационных схем мы ограничиваем себя правилами комбинационной композиции.

На **рис. 2.6 (b)** показаны два блока комбинационной логики с произвольным числом выходов одного блока, которые являются входами для другого блока.

Правила *комбинационной композиции* говорят нам, как мы можем построить большую комбинационную схему из более маленьких комбинационных элементов. Схема является комбинационной, если она состоит из соединенных между собой элементов и выполнены следующие условия:

- каждый элемент схемы сам является комбинационным;
- каждое соединение схемы является или входом, или подсоединено к одному-единственному выходу другого элемента схемы;
- схема не содержит циклических путей: каждый путь в схеме проходит через любое соединение не более одного раза.

Пример 2.1 КОМБИНАЦИОННЫЕ СХЕМЫ

Какие из схем на **рис. 2.7** являются, согласно правилам комбинационной композиции, комбинационными?

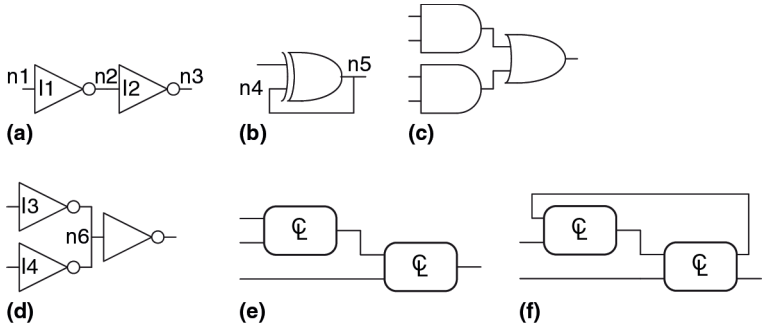


Рис. 2.7 Примеры схем

Решение Схема (a) – комбинационная. Она состоит из двух комбинационных элементов (инверторы I1 и I2). В ней три соединения: n1, n2 и n3. Соединение n1 – вход схемы и вход для I1; n2 – внутреннее соединение, являющееся выходом для I1 и входом для I2; n3 – выход схемы и выход I2. Схема (b) – это не комбинационная схема, поскольку в ней есть циклический путь: выход элемента «Исключающее ИЛИ» подключен к одному из его собственных входов, то есть циклический путь, начинаясь в n4, проходит через «Исключающее ИЛИ» к n5, который ведет обратно к n4. Схема (c) – комбинационная, а (d) – не комбинационная, поскольку соединение n6 подключено к выходам двух элементов (I3 и I4). Схема (e) – комбинационная, представляющая собой две комбинационные схемы, соединенные между собой и образующие более крупную комбинационную

схему. Схема (f) не отвечает правилам комбинационной композиции, поскольку в ней есть циклический путь через два элемента. В зависимости от функций этих элементов эта схема может быть, а может и не быть комбинационной.

Большие схемы, такие как микропроцессоры, могут быть очень сложными, поэтому мы будем применять принципы, описанные в **главе 1**, чтобы справиться со сложностью. Рассмотрение схемы как «черного ящика» с тщательно определенными интерфейсом и функцией является результатом применения принципов абстракции и модульности. Построение схемы из более мелких элементов является применением иерархического подхода к разработке. Использование правил комбинационной композиции означает применение дисциплины.

Функциональная спецификация комбинационной схемы обычно задается в виде таблицы истинности или логической функции. В следующих разделах будет описано, как вывести логическую функцию из любой таблицы истинности и как применять булеву алгебру и карты Карно для упрощения уравнений. Мы рассмотрим, как реализовывать эти уравнения, используя логические элементы, и как анализировать скорость работы таких схем.

2.2. Логические функции

Логические функции используют переменные, имеющие значение ИСТИНА или ЛОЖЬ, поэтому они идеально подходят для описания цифровой логики. В этом разделе сначала будет приведена терминология, часто используемая в логических функциях, а затем будет показано, как записать логическое выражение для любой логической функции по ее таблице истинности.

2.2.1. Терминология

Дополнение (complement) переменной A – это ее отрицание $\overline{A}$. Переменная или ее дополнение называется *литералом*. Например, A и $\overline{A}$, B и $\overline{B}$ – литералы. Мы будем называть A прямой формой переменной, а $\overline{A}$ – комплементарной формой; «прямая форма» не подразумевает, что значение A равно ИСТИНЕ, а говорит лишь о том, что у A нет черты сверху.

Операция «И» над одним или несколькими литералами называется *конъюнкцией*, *произведением* (product) или *импликантой*. $\overline{A}B$, $A\overline{B}\overline{C}$ и B являются импликантами для функции трех переменных. *Минтерм* (minterm, элементарная конъюнктивная форма) – это произведение, включающее все входы функции. $A\overline{B}\overline{C}$ – это минтерм для функции трех переменных A, B и C, а $\overline{A}B$ – не минтерм, поскольку он не включает в себя C. Аналогично операция ИЛИ над одним или более литералами на-

зывается *дизъюнкцией*, или суммой. *Макстерм* (maxterm, элементарная дизъюнктивная форма) – это сумма всех входов функции. $A + \overline{B} + C$ является макстермом функции трех переменных A, B и C.

Порядок операций важен при анализе логических функций. Означает ли $Y = A + BC$, что $Y = (A$ ИЛИ $B)$ И C или $Y = A$ ИЛИ $(B$ И $C)$? В логических функциях наибольший приоритет имеет операция НЕ, затем идет И, потом ИЛИ. Как и в обычных уравнениях, произведения вычисляются до вычисления сумм. Таким образом, правильно уравнение читается как $Y = A$ ИЛИ $(B$ И $C)$. Выражение **(2.1)** – еще один пример, показывающий порядок операций.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})). \qquad (2.1)$$

2.2.2. Дизъюнктивная форма

Таблица истинности для функции N переменных содержит 2^N строк, по одной для каждой возможной комбинации значений входов. Каждой строке в таблице истинности соответствует минтерм, который имеет значение ИСТИНА для этой строки. На **рис. 2.8** показана таблица истинности функции двух переменных A и B. В каждой строке показан соответствующий ей минтерм. Например, минтерм для первой строки – это $\overline{A}\overline{B}$, поскольку $\overline{A}\overline{B}$ имеет значение ИСТИНА тогда, когда $A = 0$ и $B = 0$. Минтермы нумеруют начиная с 0; первая строка соответствует минтерму 0 (m_0), следующая строка – минтерму 1 (m_1) и т. д.

A	B	Y	минтерм	обозначение минтерма
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	0	$A\ B$	m_3

Рис. 2.8 Таблица истинности и минтермы

Можно описать логическую функцию для любой таблицы истинности путем суммирования всех тех минтермов, для которых выход Y имеет значение ИСТИНА. Например, на **рис. 2.8** есть только одна строка (минтерм), для которой выход Y имеет значение ИСТИНА, она отмечена синим цветом. Таким образом, $Y = \overline{A}B$. На **рис. 2.9** показана таблица, в которой выход имеет значение ИСТИНА для нескольких строк. Суммирование отмеченных минтермов дает $Y = \overline{A}B + AB$.

Такая сумма минтермов называется *совершенной дизъюнктивной нормальной формой* функции (sum-of-products canonical form). Она представляет собой сумму (операцию ИЛИ) произведений (операций И, образующих минтермы). Хотя существует много способов записать одну и ту же функцию, такую как $Y = \overline{A}B + AB$, мы будем записывать минтермы в том же порядке, как в таблице истинности, чтобы всегда получать одно и то же логическое выражение для одной и той же таблицы истинно-

сти. Совершенная дизъюнктивная нормальная форма также может быть записана через символ суммы Σ. При использовании такого обозначения функция на **рис. 2.9** будет выглядеть так:

$$F(A, B) = \Sigma(m_1, m_3)$$

или (2.2)

$$F(A, B) = \Sigma(1, 3).$$

A	B	Y	минтерм	обозначение минтерма
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

Рис. 2.9 Таблица истинности с несколькими минтермами, равными ИСТИНЕ

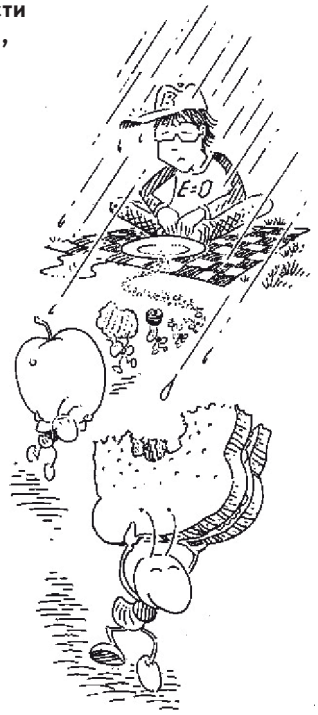
Пример 2.2 ДИЗЪЮНКТИВНАЯ ФОРМА

У Бена Битдидла намечается пикник. Он не обрадуется, если пойдет дождь или появятся муравьи. Постройте схему, в которой выход будет принимать значение ИСТИНА только в том случае, если Бену пикник понравится.

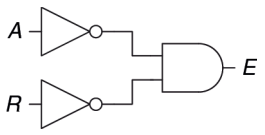
Решение Сначала определим входы и выходы. Входами будут переменные A и R, что означает муравьев (ants) и дождь (rain). Значение A принимает значение ИСТИНА, когда муравьи есть, и ЛОЖЬ, когда муравьев нет. Аналогично R имеет значение ИСТИНА, когда идет дождь, и ЛОЖЬ, когда светит солнце. Выход E (enjoyment, радость) показывает настроение Бена. E имеет значение ИСТИНА, когда Бен радуется пикнику, и ЛОЖЬ, когда он грустит. На **рис. 2.10** показана таблица истинности впечатлений Бена от пикника.

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

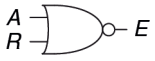
Рис. 2.10 Таблица истинности Бена



Используя дизъюнктивную форму, запишем уравнение так: $E = \overline{A}\ \overline{B}$ или $E = \Sigma(0)$. Мы можем реализовать соответствующую схему, используя два инвертора и двухвходовый элемент И, как показано на **рис. 2.11 (a)**. Вы могли заметить, что эта таблица является точно такой же, как и таблица для функции ИЛИ-НЕ, рассмотренной в **разделе 1.5.5**: $E = A$ ИЛИ-НЕ $R = \overline{A+R}$. На **рис. 2.11 (b)** показана реализация логической функции с помощью элемента ИЛИ-НЕ. В **разделе 2.3** мы покажем, что выражения $\overline{A}\overline{R}$ и $\overline{A+R}$ эквивалентны.



(a)



(b)

Рис. 2.11 Комбинационная схема Бена

Совершенная дизъюнктивная нормальная форма позволяет записать логическое выражение для любой таблицы

истинности с любым количеством переменных. На **рис. 2.12** показана произвольная таблица истинности для трехвходового элемента. Совершенная дизъюнктивная нормальная форма соответствующей логической функции выглядит так:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,C,$$
$$\text{или} \qquad (2.3)$$
$$Y = \Sigma(0, 4, 5).$$

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Рис. 2.12 Произвольная таблица истинности с тремя входами

К сожалению, совершенная дизъюнктивная нормальная форма не всегда позволяет получить простое выражение. В **разделе 2.3** мы покажем, как записать одну и ту же функцию, используя меньшее число членов логического выражения.

2.2.3. Конъюнктивная форма

Альтернативный способ выражения логических функций – это *совершенная конъюнктивная нормальная форма* (products-of-sum forms). Каждая строка таблицы истинности соответствует макстерму, который имеет значение ЛОЖЬ для этой строки. Например, макстерм для первой строки для двухвходовой таблицы истинности – это $(A + B)$, поскольку $(A + B)$ имеет значение ЛОЖЬ, когда $A = 0$ и $B = 0$. Для любой схемы, заданной таблицей истинности, мы можем записать ее логическую функцию как логическое И всех макстермов, для которых выход имеет значение ЛОЖЬ. Совершенная конъюнктивная нормальная форма также может быть записана с использованием символа П.

A	B	Y	макстерм	обозначение макстерма
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Рис. 2.13 Таблица истинности с макстермами

Пример 2.3 КОНЪЮНКТИВНАЯ ФОРМА

Запишите уравнение в совершенной конъюнктивной нормальной форме для таблицы истинности на **рис. 2.13**.

Решение Таблица истинности имеет две строки, в которых выход имеет значение ЛОЖЬ. Следовательно, функция может быть записана в конъюнктивной форме так: $Y = (A + B)\ (\overline{A} + B)$. Также функция может быть записана как $Y = \Pi(M0, M2)$, или $Y = \Pi(0, 2)$. Первый макстерм, $(A + B)$, гарантирует, что $Y = 0$ для $A = 0$ и $B = 0$, так как логическое «И» любого значения и нуля дает ноль. Аналогично второй макстерм $(\overline{A} + B)$ гарантирует, что $Y = 0$ для комбинации $A = 1$ и $B = 0$. На **рис. 2.13** показана такая же таблица истинности, как и на **рис. 2.9**, чтобы продемонстрировать, что одна и та же функция может быть записана несколькими способами.

Аналогично логическое выражение для пикника Бена (**рис. 2.10**) может быть записано в совершенной конъюнктивной нормальной форме, если обвести три строки с нулями, для того чтобы получить

$$E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R}), \text{ или } E = (1, 2, 3).$$

Это не такая красивая запись, как дизъюнктивное уравнение, $E = \overline{B}\,\overline{R}$, но эти два уравнения логически эквивалентны. Дизъюнктивная форма дает более короткое выражение, когда выход имеет значение ИСТИНА только в нескольких строках таблицы истинности; конъюнктивная же форма проще, когда выход имеет значение ЛОЖЬ только в нескольких строках таблицы истинности.

2.3. Булева алгебра

В предыдущем разделе мы изучили, как записывать логические выражения при наличии таблицы истинности. Но выражение, получаемое таким способом, не обязательно приводит к минимальному набору логических элементов. Вы можете использовать *булеву алгебру* для упрощения логических выражений точно так же, как используете алгебру для упрощения математических выражений. Правила булевой алгебры очень похожи на правила обычной алгебры, но в некоторых случаях они проще, потому что переменные могут принимать только два возможных значения: 0 или 1.

Булева алгебра основана на наборе аксиом, которые мы считаем верными. Аксиомы являются недоказуемыми в том смысле, что определение не может быть доказано. С помощью этих аксиом мы доказываем все теоремы булевой алгебры.

Эти теоремы имеют огромную практическую значимость, потому что с их помощью мы учимся тому, как упрощать логические уравнения, чтобы получать более дешевые и компактные схемы. Аксиомы и теоремы булевой алгебры подчиняются принципу двойственности. Если взаимно заменить символы 0 и 1, а также взаимно заменить операторы $\cdot$ (И) и + (ИЛИ), то логическое выражение останется верным. Мы используем символ «штрих» (′) для обозначения *двойственного* выражения.

2.3.1. Аксиомы

В **табл. 2.1** приведены аксиомы булевой алгебры. Эти пять аксиом и двойственные им аксиомы определяют логические переменные и значения операторов НЕ, И, ИЛИ. Аксиома A1 показывает, что логическая переменная B имеет значение 0, если она не имеет значение 1. Двойственное выражение для этой аксиомы A1′ утверждает, что переменная принимает значение 1, если она не имеет значение 0. Вместе аксиомы A1 и A1′ говорят нам, что мы работает в булевом, то есть бинарном, поле, состоящем из значений нулей и единиц. Аксиомы A2 и A2′ определяют операцию НЕ. Аксиомы с A3 по A5 определяют операцию И, а их двойственные аксиомы (A3′–A5′) – операцию ИЛИ.

Таблица 2.1 Аксиомы булевой алгебры

Аксиома		Двойственная аксиома		Название
A1	$B = 0$, если $B \neq 1$	A1′	$B = 1$, если $B \neq 0$	Бинарное поле
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	НЕ
A3	$0 \cdot 0 = 0$	A3′	$1 + 1 = 1$	И/ИЛИ
A4	$1 \cdot 1 = 1$	A4′	$0 + 0 = 0$	И/ИЛИ
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	И/ИЛИ

2.3.2. Теоремы одной переменной

Теоремы с T1 по T5 в **табл. 2.2** описывают, как упростить уравнения, содержащие одну переменную.

Теорема *идентичности* T1 утверждает, что для любой логической переменной B выполняется соотношение B И 1 = B. Двойственная ей теорема говорит о том, что B ИЛИ 0 = B. В аппаратуре, как показано на **рис. 2.14**, T1 означает, что если уровень сигнала на одном из входов двухвходового элемента И всегда равен 1, то мы можем удалить этот элемент и заменить его проводом, соединяющим выход этого элемента с входом B, значение которого может меняться. Точно так же теорема T1′ говорит о том, что если один вход двухвходового элемента ИЛИ всегда равен 0, мы можем заменить этот элемент на провод, соединенный с входом B. Как правило, элементы имеют определенную стоимость, энергопотребление и задержку прохождения сигнала, поэтому замена элемента на провод является целесообразной.

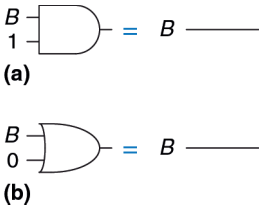


Рис. 2.14 Теорема идентичности в аппаратуре: (a) T1, (b) T1′

Таблица 2.2 Теоремы булевой алгебры для одной переменной

Теорема		Двойственная теорема		Название
T1	$B \cdot 1 = B$	T1′	$B + 0 = B$	Идентичность
T2	$B \cdot 0 = 0$	T2′	$B + 1 = 1$	Нулевой элемент
T3	$B \cdot B = B$	T3′	$B + B = B$	Идемпотентность
T4			$\overline{\overline{B}} = B$	Инволюция
T5	$B \cdot \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Дополнительность

Теорема о *нулевом элементе* T2 говорит, что B И 0 всегда равно 0. Следовательно, 0 называют нулевым элементом для операции И, потому что он обнуляет эффект любого другого входа. Двойственная ей теорема говорит о том, что B ИЛИ 1 всегда равно 1. Таким образом, 1 – это нулевой элемент для операции ИЛИ. В аппаратуре, как показано на **рис. 2.15**, если один вход элемента И равен 0, мы можем заменить элемент И проводом, подключенным к низкому логическому уровню (0). Точно так же, если один из входов элемента ИЛИ равен 1, мы можем заменить элемент ИЛИ на провод, который подключен к высокому логическому уровню (1).

Теорема об *идемпотентности* T3 утверждает, что операция логического И двух равных друг другу переменных имеет значение, равное этой переменной. Аналогичное утверждение верно для операции ИЛИ с двумя одинаковыми значениями на входах. Название теоремы происходит от латинских слов «idem» – *тот же, такой же* и «potent» – *сила*. Операции возвращают те же значения, которые вы подаете им на вход. На **рис. 2.16** показано, как идемпотентность позволяет заменить элемент схемы на провод.

Теорема об *инволюции* T4 – это забавный способ описания того, что двойное отрицание переменной дает ее исходное значение. Два последовательно включенных инвертора логически отменяют друг друга, то есть они эквивалентны проводу, как показано на **рис. 2.17**. Двойственной ей теоремой является она сама.

Теорема о *дополнительности* T5 (**рис. 2.18**) утверждает, что операция И над переменной и ее инверсным значением дает 0 (потому что одна из них всегда будет равна нулю). И согласно принципу двойственности, операция ИЛИ над переменной и ее инверсным значением всегда дает 1 (так как одна из них всегда будет равна единице).

Теорема о нулевом элементе приводит к нелепым утверждениям, которые при этом оказываются верными! Эта теорема становится особенно опасной, когда ее применяют те, кто делает рекламу: «ВЫ ПОЛУЧИТЕ МИЛЛИОН ДОЛЛАРОВ, или мы пришлем вам по почте зубную щетку» (скорее всего, вы получите зубную щетку по почте).

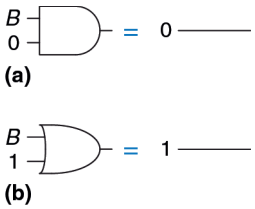


Рис. 2.15 Теорема о нулевом элементе в аппаратуре: (a) T2, (b) T2′

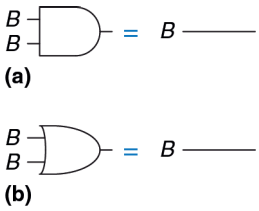


Рис. 2.16 Теорема об идемпотентности в аппаратуре: (a) T3, (b) T3′

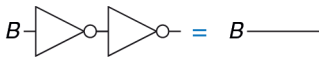


Рис. 2.17 Теорема об инволюции в аппаратуре: T4′

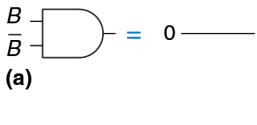


Рис. 2.18 Теорема о дополнительности в аппаратуре: (a) T5, (b) T5′

2.3.3. Теоремы с несколькими переменными

Теоремы с T6 по T12 в **табл. 2.3** описывают, как упростить уравнения, включающие в себя более одной булевой переменной.

Теоремы T6 о *коммутативности* и T7 об *ассоциативности* работают так же, как и в традиционной алгебре. В соответствии с принципом коммутативности порядок входов для функций И или ИЛИ не влияет на значение выхода. Согласно принципу ассоциативности любое группирование входов не влияет на значение выхода.

Теорема о *дистрибутивности* T8 является точно такой же, как и в традиционной алгебре, а двойственная ей теорема T8′ – нет. Согласно теореме T8 оператор И дистрибутивен относительно операции ИЛИ. T8′ говорит, что оператор ИЛИ дистрибутивен относительно операции И. В традиционной алгебре оператор умножения дистрибутивен относительно операции сложения, но не наоборот, то есть

$$(B + C)\times(B + D) \neq B + (C\times D).$$

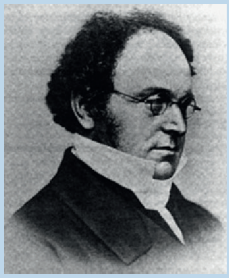
Теоремы *поглощения*, *склеивания* и *согласованности* T9–T11 позволяют нам удалять лишние переменные.

Таблица 2.3 Теоремы булевой алгебры для нескольких переменных

Теорема		Двойственная теорема		Название
T6	$B\cdot C = C\cdot B$	T6′	$B + C = C + B$	Коммутативность
T7	$(B\cdot C)\cdot D = B\cdot(C\cdot D)$	T7′	$(B + C) + D = B + (C + D)$	Ассоциативность
T8	$(B\cdot C) + (B\cdot D) = B\cdot(C + D)$	T8′	$(B + C)\cdot(B + D) = B + (C\cdot D)$	Дистрибутивность
T9	$B\cdot(B + C) = B$	T9′	$B + (B\cdot C) = B$	Поглощение
T10	$(B\cdot C) + (B\cdot \overline{C}) = B$	T10′	$(B + C)\cdot(B + \overline{C}) = B$	Склеивание
T11	$(B\cdot C) + (\overline{B}\cdot D) + (C\cdot D) = B\cdot C + \overline{B}\cdot D$	T11′	$(B + C)\cdot(\overline{B} + D)\cdot(C + D) = (B + C)\cdot(\overline{B} + D)$	Согласованность
T12	$\overline{B_0}\cdot\overline{B_1}\cdot\overline{B_2}\ldots = (\overline{B_0} + \overline{B_1} + \overline{B_2}\ldots)$	T12′	$\overline{B_0} + \overline{B_1} + \overline{B_2}\ldots = (\overline{B_0}\cdot\overline{B_1}\cdot\overline{B_2}\ldots)$	Теорема де Моргана

Теорема *де Моргана* T12 является очень важным инструментом при разработке цифровых устройств. Эта теорема утверждает, что дополнение результата умножения всех термов равно сумме дополнений каждого терма. Аналогично дополнение суммы всех термов равно результату умножения дополнений каждого терма.

В соответствии с теоремой де Моргана элемент И-НЕ эквивалентен элементу ИЛИ с инвертированными входами. Аналогично ИЛИ-НЕ эквивалентен элементу И с инвертированными входами. На **рис. 2.19** показаны эквивалентные по де Моргану элементы И-НЕ и ИЛИ-НЕ. Каждая пара символов, приведенная для каждой функции, называется двойственной. Они логически эквивалентны и взаимозаменяемы.



Август де Морган, умер в 1871 г. Британский математик, родился в Индии. Был слепым на один глаз. Его отец умер, когда ему было 10 лет. Поступил в Тринити-Колледж в Кембридже и был назначен профессором математики в возрасте 22 лет в только что открытом в то время Лондонском университете. Много писал на различные математические темы, включая логику, алгебру и парадоксы. В честь де Моргана был назван кратер на Луне. Он придумал загадку про год своего рождения: «Мне было X лет в году X2».

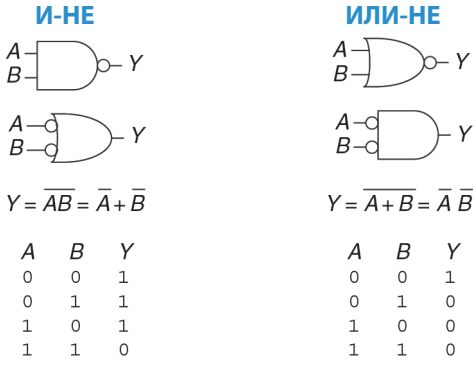


Рис. 2.19 Эквивалентные по де Моргану элементы

Кружочек на графическом обозначении элементов является обозначением отрицания (инверсии). Интуитивно вы можете представить, что если «вдавить» этот кружочек с одной стороны логического элемента, то он «выскочит» на другой, при этом тип элемента изменится с И на ИЛИ (и наоборот). Это называется «перемещением инверсии». Например, элемент И-НЕ на **рис. 2.19** состоит из элемента И с отрицанием на выходе. Перемещение инверсии влево приводит к получению элемента ИЛИ с двумя отрицаниями на входах. Базовые правила для перемещения инверсии таковы:

- перемещение инверсии назад (от выхода) или вперед (от входов) меняет тип элемента с И на ИЛИ и наоборот;
- перемещение инверсии с выхода назад ко входам приводит к тому, что на всех входах появляется инверсия;
- перемещение инверсии со всех входов элемента к выходу приводит к появлению инверсии на выходе.

В **разделе 2.5.2** принцип перемещения инверсии используется для анализа схем.

Пример 2.4 КОНЪЮНКТИВНАЯ ФОРМА ЛОГИЧЕСКОЙ ФУНКЦИИ

На **рис. 2.20** приведена таблица истинности для булевой функции Y и ее дополнения Y. Используя теорему де Моргана, получите конъюнктивную нормальную форму функции Y из дизъюнктивной формы Y.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Рис. 2.20 Таблица истинности, показывающая Y и Y'

A	B	Y	$\overline{Y}$	минтерм
0	0	0	1	$\overline{A}\,\overline{B}$
0	1	0	1	$\overline{A}\,B$
1	0	1	0	$A\,\overline{B}$
1	1	1	0	$A\,B$

Рис. 2.21 Таблица истинности, показывающая Y и Y'

Решение На **рис. 2.21** обведены минтермы, содержащиеся в функции Y. Дизъюнктивная нормальная форма функции Y имеет следующий вид:

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B. \tag{2.4}$$

Применяя операцию инверсии к обеим частям уравнения и дважды используя теорему де Моргана, получаем:

$$\overline{\overline{Y}} = (Y) = \overline{\overline{A}\,\overline{B} + \overline{A}B} = (\overline{\overline{A}\,\overline{B}})(\overline{\overline{A}B}) = (A + B)(A + \overline{B}). \tag{2.5}$$

2.3.4. Доказательство теорем булевой алгебры

Любопытный читатель может задать вопрос о том, как же доказать правильность теоремы. В булевой алгебре доказательство теорем с конечным числом переменных является простым: нужно показать, что теорема верна для всех возможных значений этих переменных. Этот метод называется *совершенной индукцией* и может быть выполнен с использованием таблицы истинности.

Пример 2.5 ДОКАЗАТЕЛЬСТВО ТЕОРЕМЫ СОГЛАСОВАННОСТИ МЕТОДОМ ПОЛНОГО ПЕРЕБОРА

Докажите теорему согласованности T11 из **табл. 2.3**.

Решение Проверьте обе части уравнения для всех восьми комбинаций переменных B, C и D. Таблица истинности на **рис. 2.22** иллюстрирует все эти комбинации. Поскольку равенство $BC + \overline{B}D + CD = BC + \overline{B}D$ верно для всех случаев, теорема доказана.

B	C	D	$BC + \overline{B}D + CD$	$BC + \overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Рис. 2.22 Таблица истинности, доказывающая теорему Т1

2.3.5. Упрощение логических уравнений

Теоремы булевой алгебры помогают нам упрощать логические уравнения. Например, возьмем дизъюнктивную форму выражения из таблицы истинности на **рис. 2.9**: $Y = \overline{A}\,\overline{B} + A\overline{B}$. В соответствии с теоремой Т10 уравнение можно упростить до $Y = B$. Это очевидно следует из таблицы истинности. В общем случае может потребоваться несколько шагов для упрощения более сложных уравнений.

Основной принцип упрощения дизъюнктивных уравнений – это комбинирование термов с использованием отношения $PA + P\overline{A} = P$, где P может быть любой импликантой. Насколько может быть упрощено логическое выражение? По определению логическое выражение в дизъюнктивной форме дизъюнктивной формы является минимизированным, если оно включает в себя минимально возможное количество импликант. Если есть несколько уравнений с одинаковым количеством импликант, минимальным будет то уравнение, в котором меньше литералов.

Импликанта называется простой (prime implicant), если она не может быть объединена с другими импликантами в уравнении, для того чтобы образовать новую импликанту с меньшим количеством литералов. Все импликанты в минимальном уравнении должны быть простыми. Иначе они могут быть объединены, чтобы уменьшить количество литералов.

Пример 2.6 МИНИМИЗАЦИЯ ЛОГИЧЕСКОЙ ФУНКЦИИ

Минимизируйте логическое выражение **(2.3)**: $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$.

Решение Мы начинаем с исходного уравнения и применяем теоремы булевой алгебры шаг за шагом, как показано в **табл. 2.4**.

Упростили ли мы полностью уравнение на этой стадии? Давайте посмотрим внимательно. В оригинальном уравнении минтермы $\overline{A}\,\overline{B}\,\overline{C}$ и $A\overline{B}\,\overline{C}$ отличаются только переменной A. Поэтому мы объединяем минтермы и получаем $\overline{B}\,\overline{C}$. Но если мы посмотрим на исходное уравнение, то заметим, что последние два минтерма $A\overline{B}\,\overline{C}$ и $A\overline{B}C$ также отличаются одним литералом (C и $\overline{C}$). Таким образом, используя тот же самый метод, мы могли бы объединить эти два минтерма и получить минтерм $A\overline{B}$. Можно сказать, что импликанты $\overline{B}\,\overline{C}$ и $A\overline{B}$ делят между собой минтерм $A\overline{B}\,\overline{C}$.

Итак, остановились ли мы на упрощении только одной пары минтермов или можем упростить обе? Используя теорему об идемпотентности, мы можем дубли-

ровать минтермы столько раз, сколько нам нужно: $B = B + B + B + B...$ Используя этот принцип, мы полностью упрощаем уравнение до его простых импликант, $\overline{B}\,\overline{C} + A\overline{B}$, как показано в **табл. 2.5**.

Таблица 2.4 Минимизация выражения

Шаг	Выражение	Объяснение
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}C$	T8: дистрибутивность
2	$\overline{B}\,\overline{C}(1) + A\overline{B}C$	T5: дополнительность
3	$\overline{B}\,\overline{C} + A\overline{B}C$	T1: идентичность

Таблица 2.5 Улучшенная минимизация выражения

Шаг	Выражение	Объяснение
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	T3: идемпотентность
2	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: дистрибутивность
3	$\overline{B}\,\overline{C}(1) + A\overline{B}(1)$	T5: дополнительность
4	$\overline{B}\,\overline{C} + A\overline{B}$	T1: идентичность

Хотя это немного нелогично, расширение импликанты (например, превращение AB в $ABC + AB\overline{C}$) иногда полезно при минимизации уравнений. Делая так, вы можете повторять один из расширенных минтермов для его объединения с другим минтермом.

Вы могли заметить, что полное упрощение булевых уравнений при помощи теорем булевой алгебры может потребовать нескольких попыток, некоторые из которых будут ошибочными. В **разделе 2.7** описана методика, позволяющая упростить процесс минимизации, – карты Карно.

Зачем же трудиться над упрощением логической функции, если оно остается логически эквивалентным? Упрощение уменьшает количество элементов, используемых при физической реализации функции в аппаратуре, тем самым делая схему меньше, дешевле и, возможно, быстрее. В следующем разделе рассказывается, как описывать логические функции при помощи логических элементов.

2.4. От логики к логическим элементам

Принципиальная схема – это изображение цифровой схемы, показывающее элементы и соединяющие их проводники. Например, схема на

рис. 2.23 показывает возможную аппаратную реализацию логической функции **(2.3)**:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C.$$

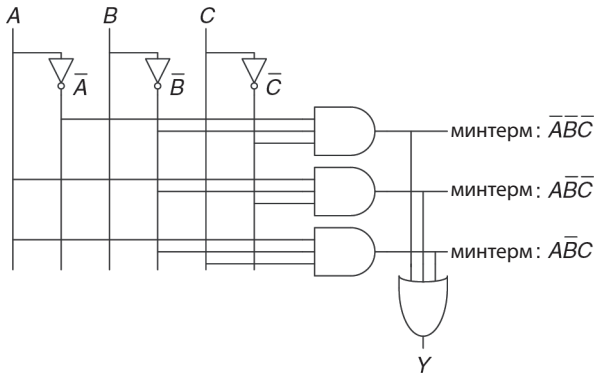


Рис. 2.23 **Схема** $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$

Изображая принципиальные схемы в унифицированном виде, нам становится легче читать и выполнять их отладку. В большинстве случаев мы будем придерживаться следующих правил:

- входы изображаются на левой (или верхней) части схемы;
- выходы изображаются на правой (или нижней) части схемы;
- всегда, когда это возможно, элементы необходимо изображать слева направо;
- проводники лучше изображать прямыми линиями, чем линиями со множеством углов (неровные рваные линии отвлекают внимание: приходится следить за тем, куда ведут провода, а не думать о том, что делает схема);
- проводники всегда должны соединяться в виде буквы Т;
- точка в месте пересечения проводников обозначает их соединение;
- проводники, пересекающиеся без точки, не имеют соединения друг с другом.

Рис. 2.24 **Способы соединения проводников**

Три последних правила показаны на **рис. 2.24**.

Любая логическая функция в дизъюнктивной форме может быть изображена в виде принципиальной схемы с использованием систематического подхода, как показано на **рис. 2.23**. Для этого надо сначала нарисовать вертикальные проводники для входов. Поместить инверторы на соседних вертикальных линиях для получения комплементарных входов, если это необходимо. Нарисовать горизонтальные линии, ведущие к элементам И, для каждого минтерма. Затем для каждого

выхода нарисовать элемент ИЛИ, соединенный с минтермом, соответствующим этому выходу. Такой стиль изображения называется программируемой логической матрицей (ПЛМ, PLA), потому что инверторы, элементы И и элементы ИЛИ систематически объединены в массивы. Программируемые логические матрицы будут рассмотрены в **разделе 5.6**.

На **рис. 2.25** показана реализация упрощенного логического выражения, которое мы получили при помощи булевой алгебры в **примере 2.6**. Заметьте, что упрощенная схема имеет значительно меньше аппаратных элементов, чем схема на **рис. 2.23**. Также ее быстродействие может быть выше, поскольку она использует элементы с меньшим количеством входов.

Мы даже можем еще уменьшить количество элементов (пусть хотя бы на один инвертор), если воспользуемся преимуществом инвертирующих логических элементов. Заметьте, что $\overline{B}\,\overline{C}$ – это элемент И с инвертированными входами. На **рис. 2.26** показана схема, которая использует эту оптимизацию для исключения инвертора на входе C. Вспомните, что согласно теореме де Моргана логический элемент И с инвертированными входами эквивалентен элементу ИЛИ-НЕ. В зависимости от технологии реализации, использование наименьшего числа элементов или использование элементов определенного типа взамен других может быть выгоднее. Например, в технологии КМОП элементы И-НЕ и ИЛИ-НЕ более предпочтительны, чем И или ИЛИ.

У многих схем имеется несколько выходов, каждый из которых вычисляет независимые логические функции для входов. Мы можем записать отдельные таблицы истинности для каждого выхода, но часто удобно записать все выходы в одну таблицу истинности и начертить одну схему для всех выходов.

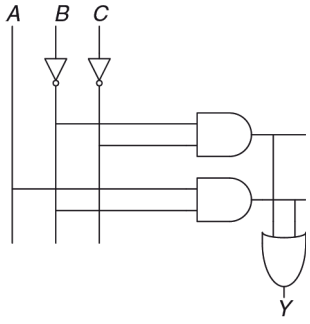


Рис. 2.25 Схема реализации функции $Y = \overline{B}\,\overline{C} + A\overline{B}$

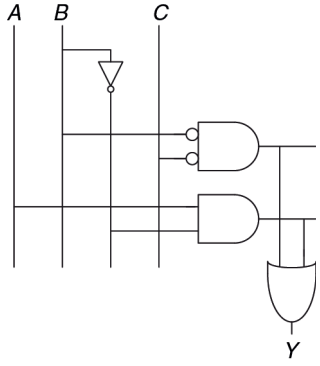


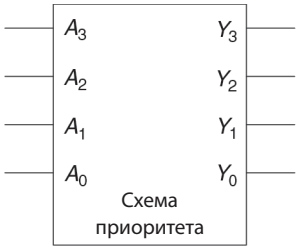
Рис. 2.26 Схема, использующая меньше элементов

Пример 2.7 СХЕМЫ С НЕСКОЛЬКИМИ ВЫХОДАМИ

Декан, заведующий кафедрой, аспирант и председатель совета общежития время от времени используют одну аудиторию. К сожалению, иногда аудитория нужна

им одновременно, что приводит к катастрофам, как, например, когда встреча декана с пожилыми и уважаемыми членами попечительского совета была запланирована на то же время, что и пивная вечеринка студентов общежития. Алиса Хакер была приглашена для того, чтобы разработать систему резервирования аудитории.

Система имеет четыре входа (A_3, ..., A_0) и четыре выхода (Y_3, ..., Y_0). Эти сигналы также могут быть записаны в виде $A_{3:0}$ и $Y_{3:0}$. Каждый пользователь активирует свой вход, когда запрашивает аудиторию на следующий день. Система активирует только один выход, подтверждая использование аудиторией самым высокоприоритетным пользователем. Декан, который оплачивает систему, требует наивысшего приоритета (3). Заведующий кафедрой, аспирант и председатель совета общежития имеют приоритеты по убыванию. Запишите таблицу истинности и логические функции для этой системы. Начертите схему, которая будет выполнять эту функцию.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Рис. 2.27 Схема приоритета

Решение Данная функция называется четырехвходовой схемой приоритета. Ее обозначение и таблица истинности приведены на **рис. 2.27**. Мы могли бы записать каждый выход в дизъюнктивной форме и упростить уравнения, используя булеву алгебру. Но достаточно посмотреть на функциональное описание (таблицу истинности), чтобы понять, каковы могут быть упрощенные уравнения: Y_3 имеет значение ИСТИНА всегда, когда подается сигнал A_3, таким образом $Y_3 = A_3$. Y_2 равен ИСТИНЕ, если подан сигнал A_2 и не подан сигнал A_3, таким образом $Y_2 = \overline{A}_3 A_2$. Y_1 имеет значение ИСТИНА, если подан сигнал A_1 и ни на какой

из более высокоприоритетных входов сигнал не подан: $Y_1 = \overline{A}_3\overline{A}_2A_1$. Y_0 имеет значение ИСТИНА при поданном сигнале A_0 и когда ни один из других выходов не активирован: $Y_1 = \overline{A}_3\overline{A}_2\overline{A}_1A_0$. Схема показана на **рис. 2.28**. Опытный разработчик часто может реализовать логическую схему, непосредственно глядя в исходные данные. При наличии четко заданной спецификации просто преобразуйте слова в уравнения, а уравнения в логические элементы схемы.

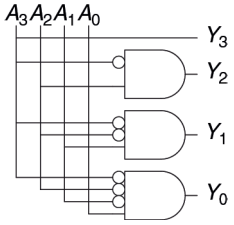


Рис. 2.28 Логическая схема

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Рис. 2.29 Таблица истинности схемы приоритета

Символ «X» используется не только для обозначения переменных, чье состояние нам безразлично, но и для обозначения недопустимых состояний сигналов при симуляции логических схем (раздел 2.6.1). Старайтесь понять из контекста, о каком варианте использования идет речь. Чтобы избежать такой двусмысленности, некоторые авторы используют символы «D» или «?» для обозначения сигналов, состояние которых нам безразлично.

Обратите внимание, что если в схеме приоритета подается сигнал A_3, то выходы схемы не будут зависеть от того, какие сигналы присутствуют на остальных входах. Мы используем символ «X» для описания состояния входов, которые нам безразличны, так как не оказывают влияния на выход. На **рис. 2.29** показано, что таблица истинности четырехвходовой приоритетной схемы становится гораздо меньше, если убрать значения входов, которыми можно пренебречь. Из этой таблицы истинности мы можем легко получить логические выражения в дизъюнктивной форме, опуская входы с X. Значения, которыми можно пренебречь, также могут возникнуть на выходах в таблице истинности, как это будет показано в **разделе 2.7.3**.

2.5. Многоуровневая комбинационная логика

Комбинационная логика, построенная как дизъюнкция конъюнкций (сумма произведений), называется двухуровневой, потому что состоит из литералов, соединенных элементами И (образуют первый уровень), выходы которых соединены с элементами ИЛИ (образуют второй уровень). Разработчики часто создают схемы с большим количеством уровней логических элементов. Такая многоуровневая комбинационная схема может использовать меньше логических элементов, чем ее двухуровневая реализация. Эквивалентные преобразования по законам де Моргана и перемещение инверсии особенно полезны при анализе и разработке многоуровневых схем.

2.5.1. Минимизация аппаратных затрат

Некоторые логические функции требуют огромного количества аппаратных ресурсов, если строить их с использованием двухуровневой логики. Показательный пример – это функция Исключающее ИЛИ (XOR) нескольких переменных. Например, рассмотрим построение трехвходового элемента XOR, используя двухуровневую технику, которую мы изучали до сих пор.

Вспомним, что *N*-входовый XOR выдает на выход значение ИСТИНА, если нечетное количество входных операндов имеют значение ИСТИНА. На **рис. 2.30 (a)** показана таблица истинности трехвходового элемента XOR. В таблице обведены строки, для которых значение выхода будет ИСТИНА. Из таблицы истинности мы понимаем форму логического выражения, соответствующую дизъюнкции конъюнкций (сумме произведений) уравнения **(2.6)**. К сожалению, это выражение невозможно упростить в меньшее количество импликант.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC. \tag{2.6}$$

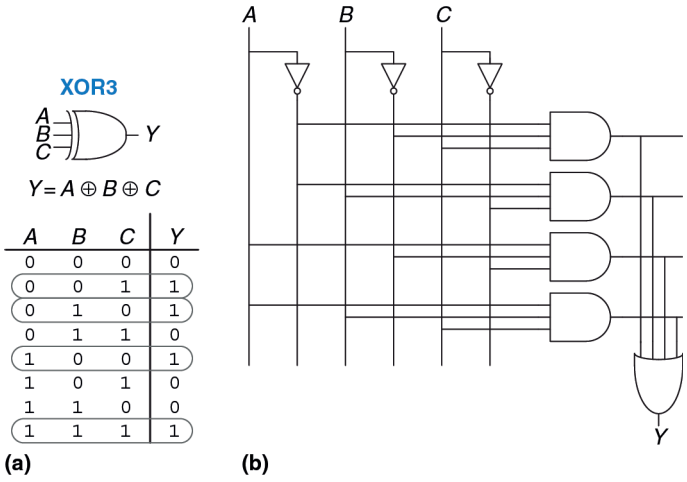


Рис. 2.30 Трехвходовый элемент XOR: функциональная спецификация (a) и реализация с двумя уровнями логики (b)

С другой стороны, A ⊕ B ⊕ C = (A ⊕ B) ⊕ C (если вы сомневаетесь, докажите это самостоятельно с помощью совершенной индукции). Следовательно, трехвходовый элемент XOR можно реализовать каскадом двухвходовых элементов XOR, как показано на **рис. 2.31**.

Аналогично восьмивходовый XOR потребует 128 восьмивходовых элементов И и одного 128-входового элемента ИЛИ для двухуровневой реализации дизъюнкции конъюнкций. Гораздо лучшей альтернативой будет использовать дерево двухвходовых элементов XOR, как показано на **рис. 2.32**.

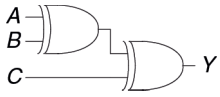


Рис. 2.31 Трехвходовый элемент XOR, собранный из двух двухвходовых элементов XOR

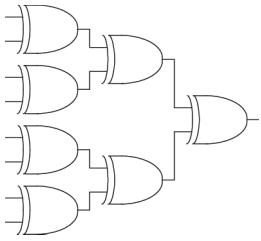


Рис. 2.32 Восьмивходовый элемент XOR, собранный из семи элементов XOR

Выбор наилучшей многоуровневой реализации заданной логической функции – это непростой процесс (выбирать наилучшую многоуровневую реализацию заданной логической функции непросто). Кроме того, понятие «наилучшее» имеет много значений: наименьшее количество элементов, лучшее быстродействие, кратчайшее время разработки, наименьшая стоимость, наименьшее энергопотребление.

В **главе 5** вы увидите, что «наилучшая» схема для одной технологии не обязательно является наилучшей для другой. Например, мы использовали элементы И и ИЛИ, но для КМОП-технологии более эффективны элементы И-НЕ и ИЛИ-НЕ. С опытом вы увидите, что для большинства схем вы сможете находить хорошую многоуровневую реализацию, просто рассматривая эти схемы (и действуя по интуиции).

Некоторый опыт вы наработаете, изучая примеры схем остальной части книги. По мере того как вы учитесь, исследуйте различные варианты разработки и думайте о компромиссах. Сейчас также доступны системы автоматизированного проектирования (САПР), которые позволяют рассматривать огромное пространство возможных многоуровневых реализаций (осуществлять поиск в многомерном пространстве решений) и находить такое, которое наилучшим образом удовлетворяет вашим критериям оптимальности с учетом имеющихся строительных блоков.

2.5.2. Перемещение инверсии

Как вы помните из **раздела 1.7.6**, для КМОП-схем лучше подходят элементы И-НЕ и ИЛИ-НЕ, а не И и ИЛИ. Но чтение уравнений многоуровневых схем с элементами И-НЕ и ИЛИ-НЕ может оказаться довольно трудным. На **рис. 2.33** показан пример многоуровневой схемы, функция которой не очевидна непосредственно из схемы. Путем перемещения инверсии можно преобразовывать подобные схемы так, что инверсия сократится, и функция может стать более понятной. Построенные на принципах из **раздела 2.3.3**, правила для перемещения инверсии таковы:

- начинать с выхода цепи и двигаться назад ко входам;

- перемещать инверсию с общего выхода на входы так, чтобы можно было читать выражение в терминах выхода (например, Y), а не инвертированного выхода $\overline{Y}$;
- продвигаясь в обратном направлении, необходимо менять каждый элемент так, чтобы число инверсий оказалось четным и их можно было сократить. Если текущий элемент имеет входные отрицания, предшествующий элемент должен быть с выходным отрицанием. Если текущий элемент не имеет входного отрицания, предшествующий должен быть без выходного отрицания.

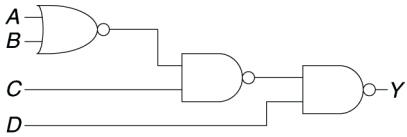


Рис. 2.33 Многоуровневая схема на элементах И-НЕ и ИЛИ-НЕ

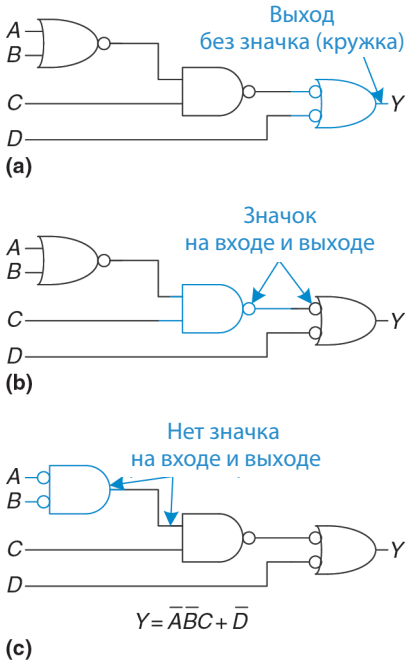


Рис. 2.34 Схема с удаленными инверсиями

Рисунок 2.34 показывает, как преобразовать схему из **рис. 2.33**, следуя изложенным правилам. Начинаем с выхода Y. Элемент И-НЕ имеет отрицание на выходе, которое мы хотим устранить. Мы переставляем выходное отрицание «назад», формируя элемент ИЛИ с инверсными входами, показанный на **рис. 2.34 (a)**. Двигаясь налево по схеме, мы замечаем, что самый правый элемент теперь имеет входное отрицание,

которое может быть отброшено вместе с выходным отрицанием среднего элемента И-НЕ так, что инверсий в этом пути не останется, как показано на **рис. 2.34 (b)**. Средний элемент не имеет входных инверсий, поэтому мы трансформируем самый левый элемент так, чтобы он не имел выходного отрицания, как показано на **рис. 2.34 (c)**. Сейчас все отрицания в схеме убраны, за исключением входов, так что функция может быть прочитана в терминах элементов И и ИЛИ с действительными или комплементарными входами: $Y = \overline{A}\,\overline{B}C + \overline{D}$.

На **рис. 2.35** показана схема, логически эквивалентная схеме на **рис. 2.34**. Функции внутренних соединений отмечены синим цветом. Поскольку следующие друг за другом отрицания могут быть отброшены, можно проигнорировать инверсии на выходе среднего и на входе самого правого элементов, получив логически эквивалентную схему на **рис. 2.35**.

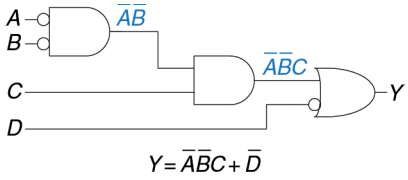


Рис. 2.35 Логически эквивалентная схема

Пример 2.8 ПЕРЕМЕЩЕНИЕ ИНВЕРСИИ В КМОП-ЛОГИКЕ

Большинство разработчиков думают в терминах элементов И и ИЛИ, но предположим, что вы хотели бы реализовать схему из **рис. 2.36** в КМОП-логике, для которой предпочтительны элементы И-НЕ и ИЛИ-НЕ. Используйте перемещение инверсии, чтобы преобразовать схему в элементы И-НЕ, ИЛИ-НЕ и НЕ.

Решение Прямолинейное решение заключается в простой замене каждого элемента И на И-НЕ с инвертором, а каждого элемента ИЛИ – на ИЛИ-НЕ с инвертором, как это показано на **рис. 2.37**. Такая схема потребует 8 элементов. Заметьте, что инверторы изображены с отрицанием на входе, а не на выходе, чтобы подчеркнуть, что двойное отрицание не меняет логику работы схемы и может быть отброшено.

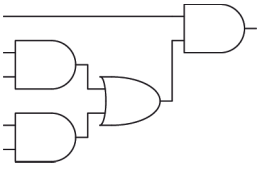


Рис. 2.36 Схема на элементах И и ИЛИ

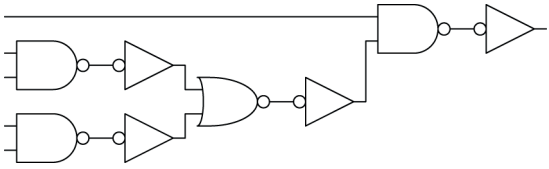


Рис. 2.37 Плохая схема на элементах И-НЕ и ИЛИ-НЕ

Обратите внимание, что отрицания могут быть добавлены на выход элемента и на вход следующего элемента без изменения функции, как показано на

рис. 2.38 (a). Выходной элемент И преобразовывается в элемент И-НЕ и инвертор, как показано на рис. 2.38 (b). Это решение требует только пяти элементов.

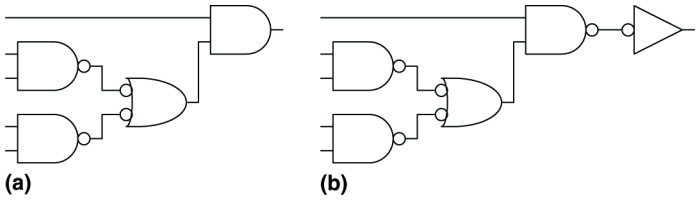


Рис. 2.38 Улучшенная схема на элементах И-НЕ и ИЛИ-НЕ

2.6. Что такое X и Z?

Булева алгебра ограничена значениями 0 и 1. Но реальные схемы могут также иметь недопустимое и отключенное состояния, представляемые символами X и Z соответственно.

2.6.1. Недопустимое значение: X

Символ «X» обозначает неизвестное логическое значение или недопустимое значение физического напряжения в соединении, не соответствующее уровням логических 0 и 1. Это обычно происходит, если к соединению подключены выходы других элементов схемы, выдающие значения 0 и 1 одновременно. На рис. 2.39 показан такой случай, когда выход *Y* подключен к элементам, имеющим на выходе ВЫСОКИЙ и НИЗКИЙ уровни.

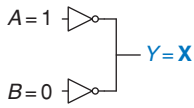


Рис. 2.39 Схема с недопустимым значением на выходе

Эта ситуация, называемая состязанием, или конфликтом (contention), считается ошибкой, и ее необходимо избегать. Реальное (физическое) напряжение на выходе с конфликтом может быть где-то между нулем и напряжением питания, в зависимости от соотношения мощностей элементов, выдающих в цепь ВЫСОКОЕ и НИЗКОЕ напряжения. Часто, но не всегда, значение напряжения оказывается в «запрещенной» зоне. Состязание также может стать причиной повышенного потребления энергии конфликтующими элементами, в результате чего схема нагревается и может быть повреждена.

Значение X также иногда используется программами моделирования для обозначения неинициализированного значения. Например, если вы забыли определить входное значение, инструмент моделирования присвоит ему значение X, для того чтобы предупредить вас о проблеме.

Как уже упоминалось в разделе 2.4, разработчики цифровых схем также используют символ «X» для обозначения в таблицах истинности безразличных переменных, от которых не зависит состояние выходов. Не путайте эти два варианта употребления символа «X». Когда X используется в таб-

лицах истинности, он показывает, что значение переменной может быть и нулем, и единицей. Когда X используется при описании схемы в схеме, это означает, что цепь имеет неизвестное или запрещенное значение.

2.6.2. Третье состояние: Z

Символ «Z» указывает, что напряжение в цепи не определяется ни источником ВЫСОКОГО, ни источником НИЗКОГО напряжения. Говорят, что такая цепь отключена, находится в состоянии высокого импеданса или в третьем состоянии. Типично неправильное представление – это что неподключенная, или находящаяся в состоянии высокого импеданса, цепь имеет значение логического 0. В реальности логическое состояние неподключенной цепи может быть как 0, так и 1, а напряжение в ней может принять некое промежуточное значение в зависимости от истории изменения состояния системы. Неподключенная цепь не обязательно означает наличие ошибки в схеме. Например, какой-нибудь другой элемент схемы может задать цепи допустимый логический уровень именно в тот момент, когда эта цепь влияет на работу схемы.

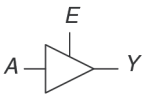
Один из распространенных способов получить неопределенное значение – это забыть подключить вход схемы к источнику напряжения логического уровня или предположить, что неподключенный вход – то же самое, что вход со значением 0. Эта ошибка может привести к тому, что поведение цепи будет случайным, так как неопределенные значения на входе могут случайно меняться из 0 в 1. Действительно, касания схемы может быть достаточно, чтобы привести к значениям сигнала из-за слабого статического электричества тела. Мы видели схему, которая корректно работала только до тех пор, пока студент держал палец на микросхеме.

Буфер с тремя состояниями, показанный на **рис. 2.40**, имеет три возможных выходных значения: ВЫСОКОЕ (1), НИЗКОЕ (0) и отключенное, или высокоимпедансное (Z), состояние[1]. Буфер с тремя состояниями имеет вход A, выход Y и сигнал управления E. Когда сигнал разрешения (управления) имеет значение ИСТИНА, буфер с тремя состояниями работает как простой буфер, передавая входное значение на выход. Когда сигнал управления имеет значение ЛОЖЬ, выход буфера переключается в третье состояние и становится плавающим (Z). Буфер с тремя состояниями на **рис. 2.40** имеет активный высокий уровень. Это значит, что когда сигнал разрешения ВЫСОКИЙ (1), передача разрешена.

На **рис. 2.41** показан буфер с тремя состояниями с активным низким уровнем. Когда сигнал управления НИЗКИЙ (0), передача разрешена. Мы видим, что сигнал имеет активный низкий уровень из-за отрицания, поставленного в его входной цепи. Мы часто обозначаем вход с активным низким уровнем, рисуя черточку (символ отрицания) над его именем ($\overline{E}$), или добавляя букву «b» или «bar» после имени, *Eb* или *Ebar*.

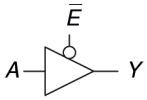
[1] Именно поэтому отключенное состояние называют третьим. – *Прим. перев.*

Буфер с третьим состоянием



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Рис. 2.40 Буфер с тремя состояниями



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Рис. 2.41 Буфер с тремя состояниями с активным низким уровнем

Буферы с третьим состоянием обычно используются в шинах, соединяющих несколько микросхем. Например, микропроцессор, видеоконтроллер и Ethernet-контроллер могут нуждаться во взаимодействии с подсистемой памяти в персональном компьютере. Каждая микросхема может подключаться к общей шине памяти, используя буферы с третьим состоянием, как показано на **рис. 2.42**. При этом только одна микросхема имеет право выставить свой сигнал разрешения, чтобы выдать значение на шину. Выходы других микросхем должны находиться в третьем состоянии, чтобы не стать причиной коллизии с микросхемой, осуществляющей обмен данными с памятью. При этом все микросхемы могут читать информацию с общей шины в любое время. Такие шины на основе буферов с тремя состояниями когда-то были очень распространенными. Но в современных компьютерах высокие скорости передачи возможны только при соединении микросхем друг с другом напрямую (point-to-point), а не с помощью общей шины.

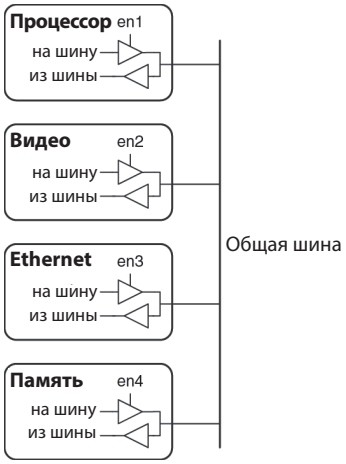


Рис. 2.42 Шина с тремя состояниями, соединяющая несколько микросхем

2.7. Карты Карно

Морис Карно родился в 1924 году. Получил степень бакалавра по физике в Городском колледже Нью-Йорка в 1948 году, а в 1952 получил степень доктора философии по физике (Ph. D., аналог степени кандидата наук) в Йельском университете.

С 1952 по 1993 год работал в Bell Labs и IBM. С 1980 по 1999 год являлся профессором информатики в Политехническом университете Нью-Йорка.

После того как вы осуществите несколько преобразований по минимизации булевых уравнений, используя булеву алгебру, вы поймете, что без соблюдения должной аккуратности иногда можно получить решение, совершенно отличное от требуемого упрощенного уравнения. Карты Карно представляют собой наглядный метод для упрощения булевых уравнений. Они были изобретены в 1953 году Морисом Карно, телекоммуникационным инженером из фирмы Bell Labs. Карты Карно очень удобны в случаях, когда уравнение содержит до четырех переменных. Но, что более важно, они дают понимание сути при манипулировании логическими выражениями.

Как мы помним, логическая минимизация осуществляется путем склейки термов. Два терма, включающих в себя импликанту P и два логических значения некоторой переменной A, объединяются, при этом переменная A исключается. Карты Карно позволяют легко находить термы, которые можно склеить, представляя их в виде таблицы.

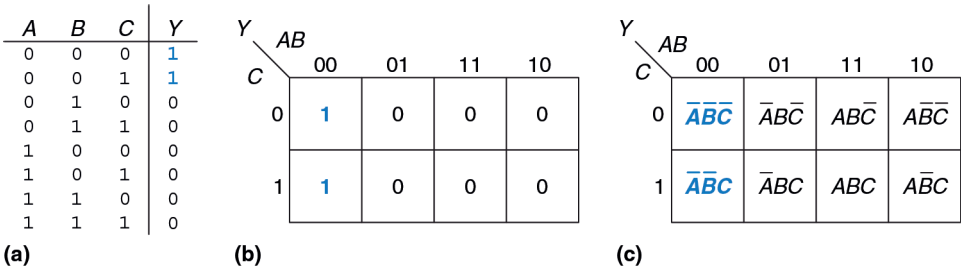


Рис. 2.43 Функция трех переменных: таблица истинности (a), карта Карно (b), карта Карно с минтермами (c)

На **рис. 2.43** показаны таблица истинности и карта Карно для функции трех переменных. Верхняя строка дает 4 возможных значения для переменных A и B. Левая колонка дает 2 возможных значения переменной C. Каждая клетка карты Карно соответствует строке таблицы истинности и содержит значение функции Y из этой строки. Например, верхняя левая клетка соответствует первой строке таблицы истинности и показывает, что значение функции Y будет равно 1, когда $ABC = 000$. Как и каждая строка в таблице истинности, каждая клетка карты Карно представляет собой отдельный минтерм. Для лучшего понимания на **рис. 2.43 (c)** показаны минтермы, соответствующие каждой клетке карты Карно.

Каждая клетка, или минтерм, отличается от соседней изменением только одной переменной. Это значит, что соседние клетки различают-

ся только в значении одного литерала, значение которого «истинно» в одной клетке и «ложно» в соседней. Например, клетки, представляющие минтермы $\overline{A}\,\overline{B}\,\overline{C}$ и $\overline{A}\,\overline{B}C$, – соседние и различаются только в переменной C. Вы, наверное, также отметили, что переменные A и B комбинируются в верхней строке в особом порядке: 00, 01, 11, 10. Этот порядок называется *кодом Грея* (Gray code). В отличие от битового порядка по возрастанию величины (00, 01, 10, 11), в коде Грея соседние записи отличаются только на один разряд. Например, 01 : 11 отличается только изменением A с 0 на 1, тогда как 01 : 10 требует изменения A из 0 в 1 и B из 1 в 0. Таким образом, обычный последовательный побитовый порядок не дает требуемого нам свойства соседних ячеек, которые должны различаться только в одной переменной.

Карты Карно также «закольцованы». Клетка с самого правого края таблицы является соседней с самой левой, так как они отличаются только в одной переменной (A). Можно свернуть карту в цилиндр, соединив края, и даже в этом случае соседние клетки так же будут отличаться только в одной переменной.

Код Грея был запатентован Фрэнком Греем, исследователем из Bell Labs, в 1953 году (патент США номер 2632058). Этот код особенно полезен для электромеханических преобразователей (например, датчиков угла поворота. – *Прим. перев.*), так как он позволяет избавиться от ложных срабатываний. Код Грея может быть любой разрядности. Например, трехбитный код Грея выглядит так:

`000, 001, 011, 010, 110, 111, 101, 100`

Льюис Кэрролл опубликовал похожую загадку в журнале Vanity Fair в 1879 году. «Правила просты. Даны два слова одинаковой длины. Нужно соединить их цепочкой слов, в которой два соседних слова отличаются лишь одной буквой», – написал он.

Например, слово SHIP можно превратить в слово DOCK так:

`SHIP, SLIP, SLOP, SLOT, SOOT, LOOT, LOOK, LOCK, DOCK.`

Можете ли вы найти более короткую цепочку?

2.7.1. Думайте об овалах

На карте Карно на **рис. 2.43** содержится только две единицы, что соответствует числу минтермов в уравнении ($\overline{A}\,\overline{B}\,\overline{C}$ и $\overline{A}\,\overline{B}\,C$). Чтение минтермов из карт Карно в точности соответствует чтению дизъюнктивной нормальной формы (ДНФ) из таблицы истинности.

Как и раньше, мы могли бы использовать булеву алгебру для минимизации:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C = \overline{A}\,\overline{B}\,(\overline{C} + C) = \overline{A}\,\overline{B}. \qquad (2.7)$$

Карты Карно помогают нам делать это упрощение графически, обводя единицы в соседних клетках овалами (n-мерными кубами), как показано на **рис. 2.44**. Для каждого овала мы пишем соответствующую ему импликанту. Вспомните из **раздела 2.2**, что импликанта является произведением одного или нескольких литералов. Переменные, для которых прямая и комплементарная формы попадают в один овал, исключаются из импликанты. В нашем случае обе формы переменной C попадают в овал, так что мы не включаем ее в импликанту. Другими словами, Y = ИСТИНА, когда $A = B = 0$ вне зависимости от C. Так что импликантой будет $\overline{A}\,\overline{B}$, карта Карно дает тот же самый ответ, какой мы получили, используя булеву алгебру.

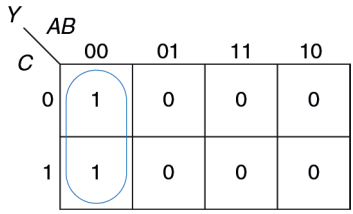


Рис. 2.44 **Минимизация при помощи карты Карно**

2.7.2. Логическая минимизация на картах Карно

Карты Карно обеспечивают простой визуальный способ минимизации логических выражений. Просто обведите все прямоугольные блоки с единицами на карте, используя наименьшее возможное число овалов. Каждый овал должен быть максимально большим. Затем прочитайте все импликанты, которые обведены.

Напомним, что формально уравнения булевой алгебры являются минимальными, только когда записаны как сумма наименьшего числа простых импликант. Каждый овал на карте Карно представляет собой импликанту. Максимально возможный овал является первичной импликантой.

Например, на карте Карно на **рис. 2.44** $\overline{A}\,\overline{B}\,\overline{C}$ и $\overline{A}\,\overline{B}\,C$ импликанты, но не первичные. На этой карте только $\overline{A}\,\overline{B}$ является первичной импликантой. Правила для нахождения минимального уравнения из карт Карно следующие:

- использовать меньше всего овалов, необходимых для покрытия всех единиц;
- все клетки в каждом овале обязаны содержать единицы;
- каждый овал должен охватывать блок, число клеток которого в каждом направлении равно степени двойки (то есть 1, 2 или 4);
- каждый овал должен быть настолько большим, насколько это возможно;
- овал может связывать края карты Карно;
- единица на карте Карно может быть обведена сколько угодно раз, если это позволяет уменьшить число овалов, которые будут использоваться.

Пример 2.9 МИНИМИЗАЦИЯ ФУНКЦИИ ТРЕХ ПЕРЕМЕННЫХ ПРИ ПОМОЩИ КАРТЫ КАРНО

Предположим, у нас есть функция $Y = F(A, B, C)$ с картой Карно, показанной на **рис. 2.45**. Упростим это выражение, используя карту Карно.

Решение Обведем единицы на карте Карно, используя наименьшее возможное количество овалов, как показано на **рис. 2.46**. Каждый овал на карте Карно

представляет собой первичную импликанту, а его размер кратен степени двойки (2×1 и 2×2).

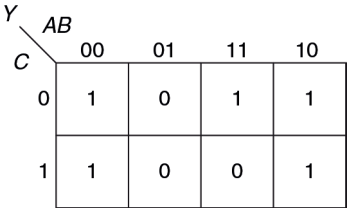


Рис. 2.45 **Карта Карно для примера 2.9**

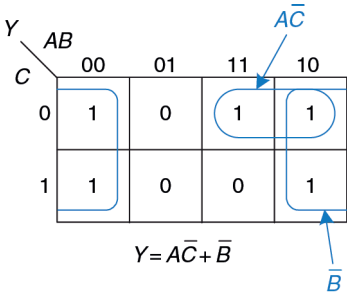


Рис. 2.46 **Решение примера 2.9**

Мы сформируем первичную импликанту для каждого выделенного овала, выписывая только те переменные, которые появляются в нем лишь в прямой или в комплементарной форме. Например, овал размером 2×1 включает в себя прямую и комплементарную формы переменной B, так что мы не включаем B в первичную импликанту. Но в этом овале есть только прямая форма переменной $A(\overline{A})$ и комплементарная форма переменной $C(\overline{C})$, так что мы включаем эти переменные в первичную импликанту $A(\overline{A})$. Подобным же образом овал размером 2×2 покрывает все клетки, где B = 0, так что первичная импликанта будет $\overline{B}$.

Обратите внимание, что правая верхняя клетка (минтерм) используется дважды, чтобы сделать овалы первичных импликант как можно большими. Как мы видели в булевой алгебре, это эквивалентно совместному использованию минтерма для уменьшения размера импликанты. Также обратите внимание на то, что овал, покрывающий четыре клетки, оборачивается через края карты Карно.

Пример 2.10 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА

Дешифратор семисегментного индикатора получает на вход четырехбитные данные $D[3:0]$ и формирует семь выходов для управления светодиодами для отображения цифр от 0 до 9. Семь выходов часто называют сегментами от a до g, или S_a–S_g, как представлено на **рис. 2.47**. Сами цифры показаны на **рис. 2.48**. Составим таблицу истинности для выходов и используем карты Карно для нахождения логического уравнения для выходов S_a и S_b. При этом предположим, что запрещенные входные значения (10–15) ничего не выводят на индикатор.

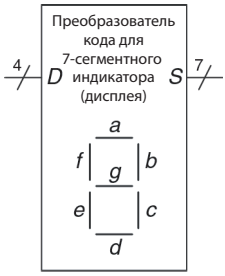


Рис. 2.47 **Семисегментный индикатор**

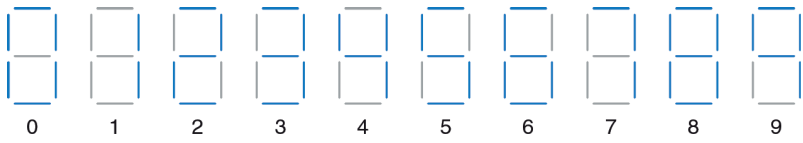


Рис. 2.48 Цифры на семисегментном индикаторе

Решение Таблица истинности дана в **табл. 2.6**. Например, вход 0000 должен включать все сегменты, за исключением S_g.

Таблица 2.6 Таблица истинности дешифратора семисегментного индикатора

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
Прочие	0	0	0	0	0	0	0

Каждый из семи выходов является независимой функцией от четырех переменных. Карты Карно для выходов S_a и S_b показаны на **рис. 2.49**. Помните, что соседние клетки могут отличаться только одной переменной, так что мы промаркируем строки и столбцы в коде Грея: 00, 01, 11, 10. Будьте осторожны и помните этот порядок, когда будете вписывать значения выходов в клетки.

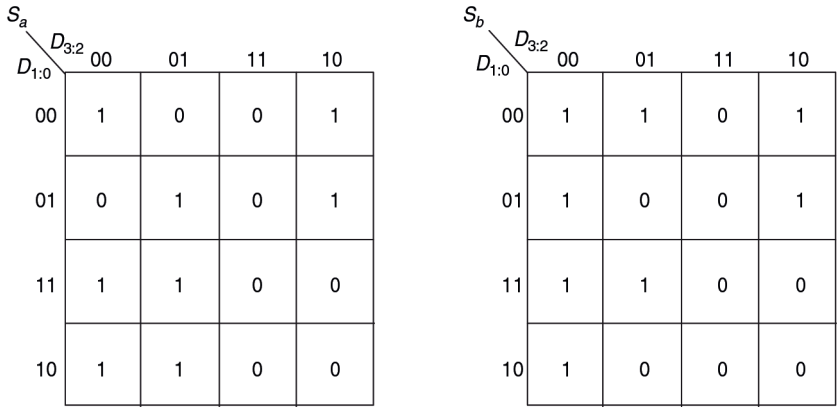


Рис. 2.49 Карты Карно для S_a и S_b

Затем обведем первичные импликанты. При этом используем минимально необходимое количество овалов для покрытия всех единиц. Овалы могут связывать края (вертикальные и горизонтальные), а каждая единица может быть выделена несколько раз. На **рис. 2.50** показаны первичные импликанты и упрощенные логические уравнения.

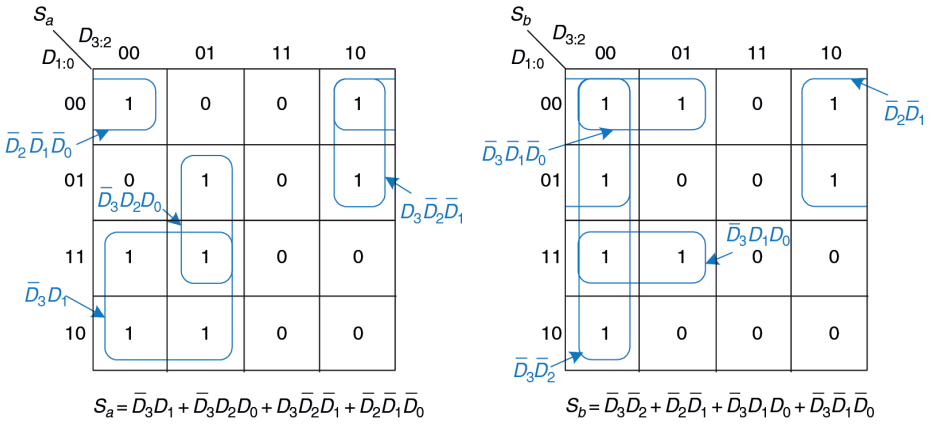


Рис. 2.50 Решение упражнения 2.10

Заметьте, что минимальный набор первичных импликант – не единственно возможный. Например, запись 0000 на карте Карно для S_a может быть выделена вместе с записью 1000, получая минтерм $\overline{D}_2\overline{D}_1\overline{D}_0$. Но вместо этого овал может включать в себя запись 0010, получая минтерм $\overline{D}_3\overline{D}_2\overline{D}_0$, как показано пунктирной линией на **рис. 2.51**.

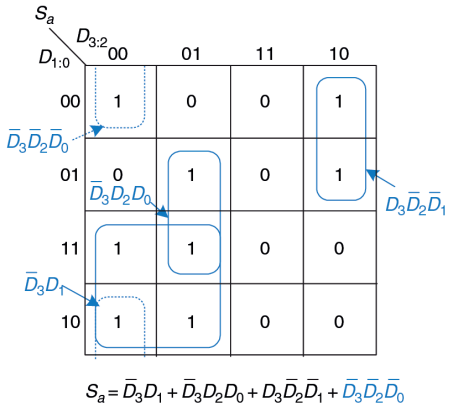


Рис. 2.51 Альтернативная карта Карно для S_a, использующая другой набор первичных импликант

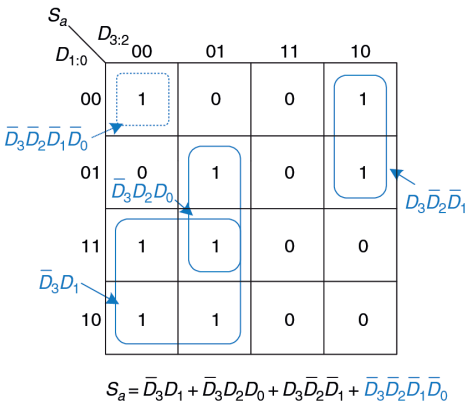


Рис. 2.52 Карта Карно для S_a, использующая некорректную импликанту

Рисунок 2.52 иллюстрирует распространенную ошибку, когда непервичная импликанта выбирается для покрытия 1 в левом верхнем углу. Этот минтерм

$\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$ дает дизъюнкцию конъюнкций (сумму произведений), которая не минимизирована. Его можно было бы скомбинировать с любым из двух соседних минтермов для получения овала большего размера, как было сделано на предыдущих двух рисунках.

2.7.3. Безразличные переменные

Вспомните, что безразличные переменные в таблице истинности были введены в **разделе 2.4** для уменьшения числа ее строк в тех случаях, когда соответствующие переменные не влияют на выход. Они обозначаются символом «X», который означает, что значение входной переменной может быть или 0, или 1.

Не только входы, но и выходы могут быть безразличными, если состояние выхода не важно или соответствующая комбинация входов никогда не возникает. Такие выходы могут трактоваться или как 0, или как 1, в зависимости от того, как решит разработчик.

В картах Карно безразличные переменные позволяют провести еще большую логическую минимизацию. Их можно включать в овалы, если это помогает покрыть единицы или меньшим количеством овалов, или овалами, большими по размеру, но их можно и не покрывать, если это не помогает минимизации.

Пример 2.11 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА С БЕЗРАЗЛИЧНЫМИ ПЕРЕМЕННЫМИ

Повторим **пример 2.10** для случая, когда нас не интересуют значения выходов при запрещенных входных значениях от 10 до 15.

Решение Карта Карно с безразличными элементами, отмеченными как «X», представлена на **рис. 2.53**. Поскольку такие элементы могут быть равны как 0, так и 1, мы используем их там, где это поможет покрыть единицы или меньшим количеством овалов, или овалами, большими по размеру. Обведенные значения X трактуются как 1, необведенные – как 0. Посмотрите, как для сегмента S_a можно выделить овал размером 2×2, объединяющий все четыре угла. Используйте клетки с безразличными значениями для упрощения логики.

2.7.4. Карты Карно: подведение итогов

Булева алгебра и карты Карно – два метода логического упрощения. В конечном счете целью является нахождение наименее затратного метода реализации конкретной логической функции.

В современной инженерной практике компьютерные программы, называемые *синтезаторами логики* (logic synthesizers), проводят упрощение схем по описанию их логических функций, как это показано в **главе 4**. Для больших задач программы логического синтеза намного

эффективнее людей. Для маленьких же задач человек с некоторым опытом может найти хорошее решение «на глаз». Никто из авторов книги тем не менее никогда не использовал карты Карно в реальной жизни для решения практических задач. Но понимание принципов, лежащих в основе карт Карно, крайне важно. Кроме того, знание карт Карно может пригодиться на собеседовании на работу в технологическую компанию!

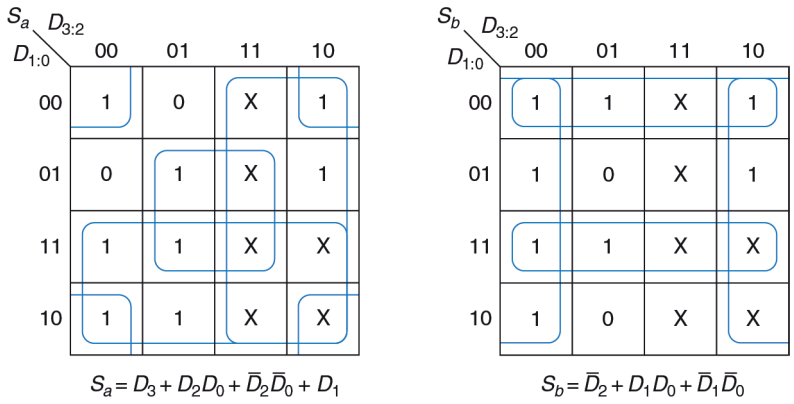


Рис. 2.53 **Карта Карно с безразличными переменными**

2.8. Базовые комбинационные блоки

Комбинационные логические элементы часто группируются в «строительные блоки», используемые для создания сложных систем. Это позволяет абстрагироваться от излишней детализации уровня логических элементов и вести разработку на уровне строительных блоков. Мы уже изучили три таких блока: полный сумматор (**раздел 2.1**), схемы приоритета (**раздел 2.4**) и дешифратор семисегментного индикатора (**раздел 2.7**). Этот раздел представляет два типа блоков, еще более часто используемых при разработке: мультиплексоры и дешифраторы. В **главе 5** будет рассказано и о других комбинационных «строительных блоках», используемых для разработки цифровых схем.

2.8.1. Мультиплексоры

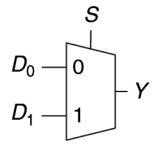
Мультиплексоры являются одними из наиболее часто используемых комбинационных схем. Они позволяют выбрать одно выходное значение из нескольких входных в зависимости от значения сигнала выбора.

Двухвходовый мультиплексор (2:1)

На **рис. 2.54** показаны условное графическое обозначение и таблица истинности для двухвходового мультиплексора (2:1) с двумя входами

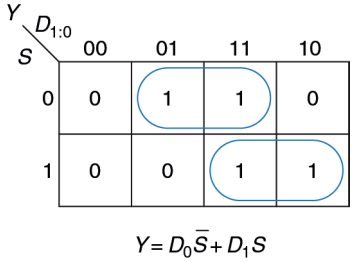
данных D_0 и D_1, входом выбора S и одним выходом Y. Мультиплексор передает на выход один из двух входных сигналов данных, основываясь на сигнале выбора: если $S = 0$, выход $Y = D_0$, и если $S = 1$, то выход $Y = D_1$. S также называют управляющим сигналом, так как он управляет поведением мультиплексора.

Двухвходовый мультиплексор может быть построен с использованием дизъюнкции конъюнкций (суммы произведений), как показано на **рис. 2.55**. Логическое выражение для него может быть получено с помощью карт Карно или составлено на основе описания ($Y = 1$, если $S = 0$ И $D_0 = 1$ ИЛИ $S = 1$ И $D_1 = 1$).



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Рис. 2.54 Условное обозначение и таблица истинности двухвходового мультиплексора



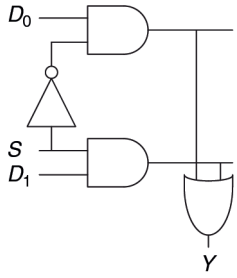


Рис. 2.55 Реализация двухвходового мультиплексора с использованием двухуровневой логики

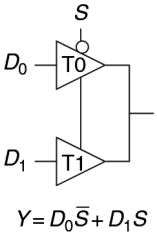


Рис. 2.56 Мультиплексор на буферах с тремя состояниями

Мультиплексор также может быть построен с помощью буферов с тремя состояниями, как показано на **рис. 2.56**. Сигналы разрешения буферов с тремя состояниями организованы так, что все время активен только один буфер. Когда $S = 0$, то включен только элемент T0, позволяющий сигналу D_0 передаваться на выход Y. Когда $S = 1$, то активен только элемент T1, передающий на выход сигнал D_1.

Многовходовые мультиплексоры

Четырехвходовый мультиплексор (4:1) имеет четыре входа данных и один выход, как показано на **рис. 2.57**. Для выбора одного из четырех входов данных требуется двухразрядный управляющий сигнал.

Четырехвходовый мультиплексор может быть построен с использованием дизъюнкции конъюнкций (суммы произведений), буферов с тремя состояниями или двухвходовых мультиплексоров, как показано на **рис. 2.58**.

Конъюнкции, подключенные к сигналам разрешения работы буферов с тремя состояниями, могут быть построены с использованием элементов И и инверторов. Они также могут быть сформированы дешифратором, который мы рассмотрим в **разделе 2.8.2**.

Мультиплексоры с большим количеством входов, например восьмивходовые или шестнадцативходовые, могут быть построены простым масштабированием. В общем случае мультиплексор N:1 требует $\log_2 N$ управляющих сигналов. Выбор наилучшей реализации, как и прежде, зависит от используемой технологии.

Строго говоря, соединение двух выходов логических элементов нарушает правила построения комбинационных схем, описанные в разделе 2.1. Но в этом конкретном случае в любой момент времени только один из этих элементов может подавать сигнал на выход Y, так что такое исключение из правил допустимо.

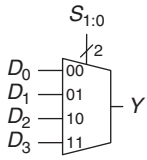


Рис. 2.57 Четырехвходовый мультиплексор

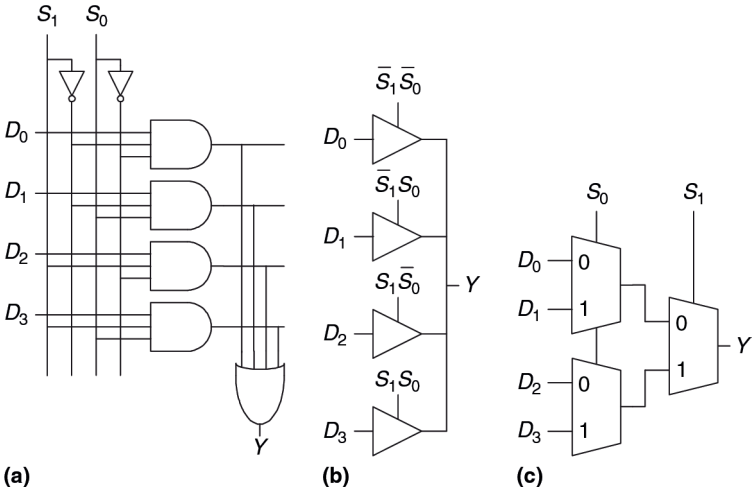


Рис. 2.58 Реализация четырехвходового мультиплексора: двухуровневая логика (a), буфер с тремя состояниями (b), иерархическая (c)

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

Логика на мультиплексорах

Мультиплексоры могут использоваться как таблицы преобразования (lookup tables) для выполнения логических функций. На **рис. 2.59** показан четырехвходовый мультиплексор, используемый для реализации двухвходового элемента И.

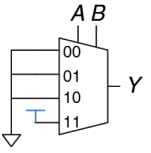


Рис. 2.59 Получение двухвходового элемента И из четырехвходового мультиплексора

Входы A и B служат управляющими линиями. Входы данных мультиплексора подключены к 0 и 1 согласно соответствующей строке таблицы истинности. Вообще, 2^N-входовый мультиплексор можно запрограммировать для выполнения любой N-входовой логической

функции, используя 0 и 1 для соответствующих входов данных. Действительно, изменением входных данных мультиплексор может быть перепрограммирован для выполнения различных функций.

Немного смекалки, и мы сможем уменьшить размер мультиплексора наполовину, используя только 2^{N-1}-входовый мультиплексор для выполнения любой N-входовой логической функции. Способ заключается в том, чтобы подавать один из литералов, так же как 0 и 1, на вход данных мультиплексора.

Для иллюстрации этого принципа на **рис. 2.60** показаны функции двухвходовых элементов И и Исключающее ИЛИ, реализованных на двухвходовых мультиплексорах. Мы начали с обычной таблицы истинности и затем скомбинировали пары строк, чтобы исключить самую правую входную переменную (B) и выразить выход в термах этой переменной. Например, в случае элемента И, когда $A = 0$, то $Y = 0$ вне зависимости от B. Когда $A = 1$, то $Y = 0$, если $B = 0$, и $Y = 1$, если $B = 1$, так что $Y = B$. Затем мы используем мультиплексор как таблицу подстановки в соответствии с новой уменьшенной таблицей истинности.

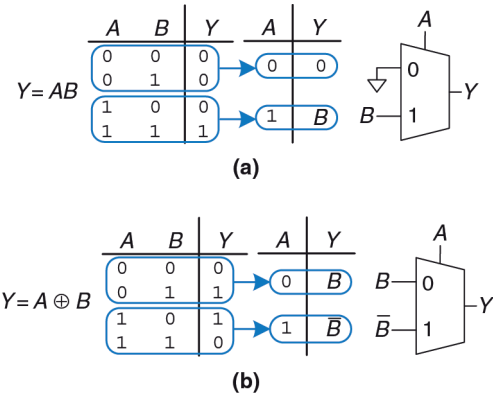


Рис. 2.60 Реализация логических функций на мультиплексорах

Пример 2.12 ЛОГИКА С МУЛЬТИПЛЕКСОРАМИ

Алисе Хакер необходимо реализовать функцию $Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$ для завершения ее курсового проекта. Когда она посмотрела, какие микросхемы доступны ей в лаборатории, то увидела, что там остался только восьмивходовый мультиплексор. Как ей реализовать эту функцию?

Решение На **рис. 2.61** показана схема, разработанная Алисой с использованием одного восьмивходового мультиплексора. Этот мультиплексор используется в качестве таблицы преобразования, где каждая строка таблицы истинности соответствует входу мультиплексора.

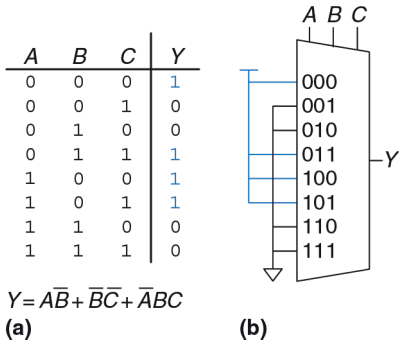


Рис. 2.61 Схема Алисы: таблица истинности (a), реализация на восьмивходовом мультиплексоре (b)

Пример 2.13 ЛОГИКА С МУЛЬТИПЛЕКСОРАМИ, ПОВТОРЕНИЕ

Алиса еще раз включила свою схему перед защитой проекта и сожгла единственный восьмивходовый мультиплексор (она случайно подала напряжение 20 В вместо 5 В после бессонной ночи).

Теперь она просит у своих друзей запасные элементы, и ей дают четырехвходовый мультиплексор и инвертор. Сможет ли она собрать свою схему, используя только эти элементы?

Решение Алиса уменьшила свою таблицу истинности до четырех строк, сделав выход зависящим от C. (Она могла бы также исключить любой из двух других столбцов таблицы истинности, сделав выход зависимым от A или B.) Новая схема показана на **рис. 2.62**.

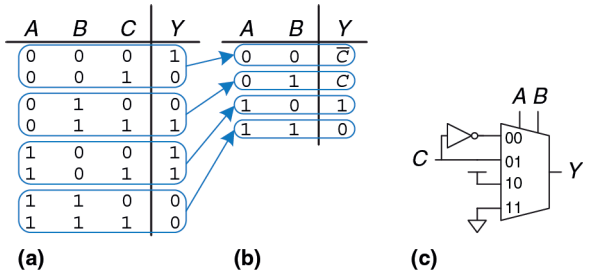


Рис. 2.62 Новая схема Алисы

2.8.2. Дешифраторы

В общем случае у дешифратора имеется N входов и 2^N выходов. Он выдает единицу строго на один из выходов в зависимости от набора входных значений. На **рис. 2.63** показан дешифратор 2:4. Когда $A[1:0] = 00$, $Y_0 = 1$. Когда $A[1:0] = 01$, $Y_1 = 1$ и т. д. Выходы образуют прямой *унитарный код* (one-hot code), называемый так потому, что в любое время только один из выходов может принимать значение единицы.

Пример 2.14 РЕАЛИЗАЦИЯ ДЕШИФРАТОРА

Реализуйте дешифратор 2:4 на элементах И, ИЛИ и НЕ.

Решение На **рис. 2.64** показана реализация дешифратора 2:4, использующая 4 элемента И. Каждый элемент зависит или от действительной, или от комплементарной формы каждого входа. Вообще, дешифратор $N{:}2^N$ может быть построен из 2^N N-входовых элементов И, к которым подходят различные комбинации действительных и комплементарных входов. Каждый выход в дешифраторе представляет собой одиночный минтерм. Например, Y_0 представляет минтерм $\overline{A}_1\overline{A}_0$. Это обстоятельство будет удобно при использовании дешифратора с другими цифровыми базовыми блоками.

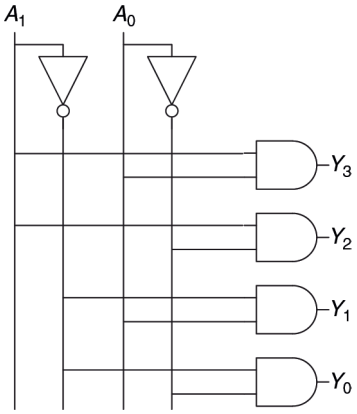


Рис. 2.64 Реализация дешифратора 2:4

Построение логических схем на дешифраторах

Дешифратор может комбинироваться с элементами ИЛИ для построения логических функций. На **рис. 2.65** показана двухвходовая функция Исключающее ИЛИ-НЕ (XNOR), использующая дешифратор 2:4 и один элемент ИЛИ. Поскольку каждый выход дешифратора представляет одиночный минтерм, функция построена как логическое ИЛИ всех минтермов этой функции. На **рис. 2.65** показана функция $Y = \overline{A}\overline{B} + AB = \overline{A \oplus B}$.

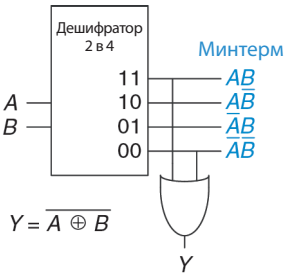


Рис. 2.65 Реализация логической функции на дешифраторе

При использовании дешифраторов для реализации логических функций проще всего выразить функцию таблицей истинности или записать ее в дизъюнктивной нормальной форме. N-входовая функция, имеющая M единиц в таблице истинности, может быть построена с использованием $N{:}2^N$ дешифратора и M-входового элемента ИЛИ, подключенных ко всем минтермам, содержащим единицу в таблице истинности. Эта идея будет применена для создания постоянного запоминающего устройства (ПЗУ) в **разделе 5.5.6**.

2.9. Временные характеристики

В предыдущих разделах мы концентрировались в первую очередь на работе схемы, в идеале использующей наименьшее число элементов. Но, как подтвердит любой опытный разработчик, одна из самых сложных задач в разработке схем – это учет всех ограничений, накладываемых на временные характеристики работы схемы, ведь хорошая схема должна работать предельно быстро и при этом без сбоев.

Изменение выходного значения в ответ на изменение входа занимает время. На **рис. 2.66** показана задержка между изменением входа буфера и последующим изменением его выхода. Этот рисунок называется *временной диаграммой*; он изображает переходную характеристику схемы буфера при изменении входа. Переход от НИЗКОГО уровня к ВЫСОКОМУ называется *передним фронтом сигнала*. Аналогично переход от ВЫСОКОГО уровня к НИЗКОМУ (на рисунке не показан) называется соответственно *задним фронтом сигнала*. Синяя стрелка показывает, что передний фронт сигнала *Y* вызывается передним фронтом сигнала *A*. Величина задержки измеряется от момента времени, когда входной сигнал *A* достигает уровня 50 %, до момента достижения уровня 50 % выходным сигналом *Y*. Уровень 50 % – это точка, в которой сигнал находится ровно посередине между НИЗКИМ и ВЫСОКИМ логическими уровнями.

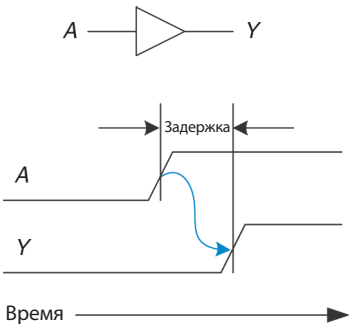


Рис. 2.66 Задержка схемы

2.9.1. Задержка распространения и задержка реакции

Комбинационная логика характеризуется *задержкой распространения* (propagation delay) и *задержкой реакции*, или отклика (contamination delay). Задержка распространения t_{pd} – это максимальное время от начала изменения входа до момента, когда все выходы достигнут

Когда разработчики говорят о задержке схемы, они в большинстве случаев имеют в виду наибольшее возможное значение задержки (задержку распространения), если только из контекста не следует другое.

установившихся значений. Задержка реакции t_{cd} – это минимальное время от момента, когда вход изменился, до момента, когда любой из выходов начнет изменять свое значение.

На **рис. 2.67** синим и серым цветами показаны соответственно задержки распространения и задержка реакции буфера. На рисунке показано, что вход A изначально имел или ВЫСОКОЕ, или НИЗКОЕ значение, и оно изменяется на противоположное в определенный момент времени; нас интересует только факт, что оно (значение A) изменилось, но не его конкретное значение. В ответ, спустя некоторое время, меняется Y. Стрелки показывают, что Y может начать меняться через временной интервал t_{cd} после изменения A и что Y точно установится в новое значение не позднее, чем через интервал t_{pd}.

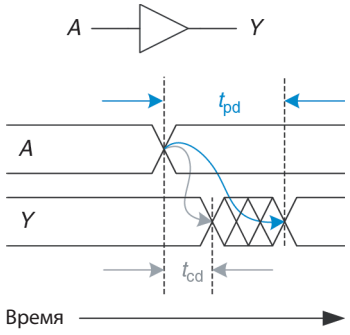


Рис. 2.67 Задержка распространения и задержка реакции

Задержки в схемах обычно составляют от нескольких пикосекунд (1 пс = 10^{-12} с) до нескольких наносекунд (1 нс = 10^{-9} с). Пока вы читали это замечание, прошло несколько триллионов пикосекунд.

Основные причины задержек в схемах заключаются во времени, требуемом для перезарядки емкостей электрических цепей, а также в конечной скорости распространения электромагнитных волн в среде. Величины t_{pd} и t_{cd} могут различаться по многим причинам, включающим в себя:

- разные задержки нарастания и спада сигнала;
- несколько входов и выходов, одни из которых быстрее, чем другие;
- замедление работы схемы при повышении температуры и ускорение при охлаждении.

Вычисление t_{pd} и t_{cd} требует рассмотрения нижних уровней абстракций, что выходит за рамки этой книги. Производители обычно предоставляют документацию со спецификацией этих задержек для каждого элемента.

Наряду с уже перечисленными факторами задержки распространения и реакции также определяются *путем*, который проходит сигнал от входа до выхода. На **рис. 2.68** показана четырехвходовая схема. *Критический путь* (critical path), выделенный синим, – это путь от входа A или B до выхода Y. Он соответствует цепи с наибольшей задержкой и является самым медленным, поскольку входному сигналу нужно пройти три эле-

мента до выхода. Этот путь критический потому, что он ограничивает скорость, с которой работает схема. Самый короткий путь в схеме, показанный серым, – путь от входа D до выхода Y. Это кратчайший и, следовательно, самый быстрый путь в схеме, т. к. входному сигналу до выхода нужно пройти только через один элемент.

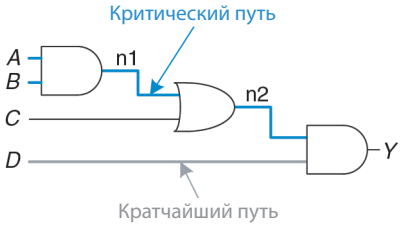


Рис. 2.68 Кратчайший путь и путь с наибольшей задержкой

Задержка распространения комбинационной схемы – это сумма задержек распространения всех элементов в критическом пути. Задержка реакции – сумма задержек реакции всех элементов в кратчайшем пути. Эти задержки показаны на **рис. 2.69** и могут быть описаны следующими уравнениями:

$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}}; \tag{2.8}$$

$$t_{cd} = t_{cd\_\mathrm{AND}}. \tag{2.9}$$

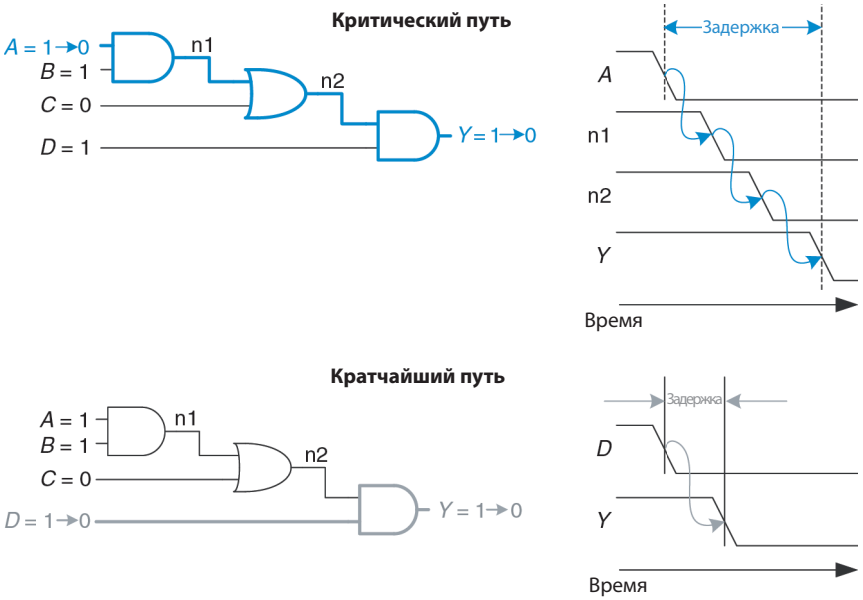


Рис. 2.69 Временные диаграммы для кратчайшего пути и пути с наибольшей задержкой

Несмотря на то что мы проигнорировали задержку распространения сигналов по проводникам, цифровые схемы в настоящее время настолько быстро работают, что эта задержка может превышать задержку в логических элементах. Связанная со скоростью света задержка распространения сигналов в проводах будет рассмотрена ниже (приложение А).

Пример 2.15 НАХОЖДЕНИЕ ЗАДЕРЖЕК

Бену надо найти задержки распространения и отклика схемы, показанной на **рис. 2.70**. Согласно справочнику каждый элемент имеет задержку распространения 100 пикосекунд (пс) и задержку отклика 60 пс.

Решение Бен начал с нахождения критического и кратчайшего путей в схеме. Критический путь, выделенный на **рис. 2.71** синим, – это путь от входа A или B через три элемента до выхода Y. Следовательно, t_{pd} – это утроенная задержка распространения для одиночного элемента, или 300 пс.

Кратчайший путь, выделенный на **рис. 2.72** серым, – это путь от входов C, D или E через два элемента до выхода Y. В кратчайшем пути только два элемента, так что t_{cd} равно 120 пс.

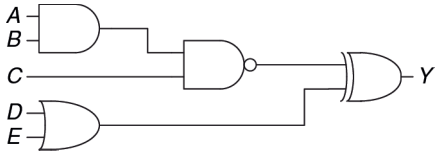


Рис. 2.70 Схема Бена

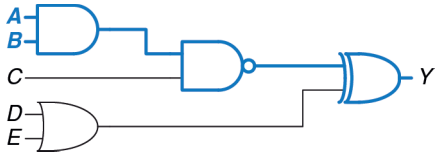


Рис. 2.71 Цепь с наибольшей задержкой

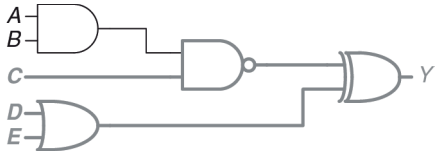
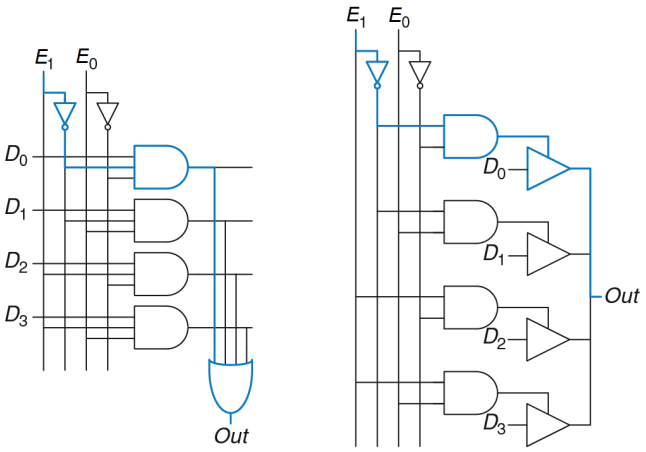


Рис. 2.72 Кратчайшая цепь

Пример 2.16 ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ МУЛЬТИПЛЕКСОРА: СРАВНЕНИЕ КРИТИЧЕСКИХ ПУТЕЙ

Сравните наихудшие временные характеристики каждой из трех реализаций четырехвходового мультиплексора, показанных на **рис. 2.58** в **разделе 2.8.1**. Задержки распространения для компонентов перечислены в **табл. 2.7**. Каким будет критический путь для каждой реализации? Исходя из анализа временных характеристик, какую схему вы предпочтете другим и почему?

Решение Один из критических путей для каждого из трех вариантов выделен синим на **рис. 2.73** и **2.74**. t_{pd_sy} показывает задержку распространения от управляющего входа S до выхода Y; t_{pd_dy} – от входа данных до выхода Y; t_{pd} – худшее из двух: $\max(t_{pd\_sy}, t_{pd\_dy})$.



$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4}$
$= 30 \text{ пс} + 80 \text{ пс} + 90 \text{ пс}$
$= \mathbf{200 \text{ пс}}$

(a)

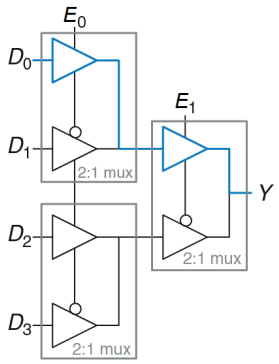
$t_{pd\_dy} = t_{pd\_AND3} + t_{pd\_OR4}$
$= \mathbf{170 \text{ пс}}$

$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_ey}$
$= 30 \text{ пс} + 60 \text{ пс} + 35 \text{ пс}$
$= \mathbf{125 \text{ пс}}$

(b)

$t_{pd\_dy} = t_{pd\_TRI\_ay}$
$= \mathbf{50 \text{ пс}}$

Рис. 2.73 Задержки распространения в четырехвходовом мультиплексоре: двухуровневая схема (a), буфер с тремя состояниями (b)



$t_{pd\_sy} = t_{pd\_TRI\_ey} + t_{pd\_TRI\_ay} = \mathbf{85 \text{ пс}}$
$t_{pd\_dy} = 2\ t_{pd\_TRI\_ay} = \mathbf{100 \text{ пс}}$

Рис. 2.74 Задержки распространения в четырехвходовом мультиплексоре, построенном из двухвходовых

Как для двухуровневой схемы, так и для реализации на буферах с тремя состояниями, на **рис. 2.73** критическим является путь от одного из сигналов управления S до выхода Y: $t_{pd} = t_{pd\_sy}$. Эта схема критическая по управлению, поскольку критический путь идет от управляющих сигналов до выхода. Любая дополнительная задержка в сигналах управления добавится непосредственно в наихудшую задержку. Задержка от D до Y на **рис. 2.73 (b)** – всего 50 пс по сравнению с задержкой от S до Y в 125 пс.

Таблица 2.7 Временные характеристики элементов в схемах мультиплексоров

Элемент	t_{pd} (пс)
НЕ	30
Двухвходовый И	60
Трехвходовый И	80
Четырехвходовый ИЛИ	90
Буфер с тремя состояниями (от *A* до *Y*)	50
Буфер с тремя состояниями (от *E* до *Y*)	35

На **рис. 2.74** показана иерархическая реализация мультиплексора 4:1, использующая два каскада мультиплексоров 2:1. Критический путь в ней от любого входа данных *D* до выхода. Эта схема критическая по данным, поскольку критический путь идет от входа данных до выхода: $t_{pd} = t_{pd\_dy}$.

Если данные приходят на входы задолго до управляющих сигналов, мы должны предпочесть схему с наименьшей задержкой от управления до выхода (иерархическая схема на **рис. 2.74**). Аналогично, если управляющие сигналы приходят намного раньше входных данных, мы должны предпочесть схему с наименьшей задержкой от данных до выхода (реализация на буферах с тремя состояниями на **рис. 2.73 (b)**).

Наилучший выбор будет зависеть не только от цепи с наибольшей задержкой, но и от потребляемой электроэнергии, стоимости и наличия компонентов.

2.9.2. Импульсные помехи

До сих по мы обсуждали случай, когда одиночное изменение входного сигнала вызывает одиночное изменение выхода. Но может оказаться, что одиночное изменение на входе вызывает несколько выходных изменений. Это называется *импульсной помехой*, или *паразитным импульсом*. Хотя паразитный импульс обычно не вызывает проблем, важно понимать, что он есть, и уметь распознавать его на временных диаграммах. На **рис. 2.75** показана схема, подверженная паразитным импульсам, и карта Карно для нее.

Логическое уравнение минимизировано корректно, но посмотрите, что происходит, когда $A = 0$, $C = 1$ и B меняется из 1 в 0. **Рисунок 2.76** иллюстрирует этот случай. Короткий путь (показан серым) проходит через два элемента: И и ИЛИ. Критический путь (показан синим) проходит через инвертор и два элемента: И и ИЛИ.

Как только B переключится из 1 в 0, n2 (в коротком пути) изменится в 0 до того, как n1 (в критическом пути) сможет установиться в 1. До установки n1 в единицу оба входа элемента ИЛИ будут принимать значение 0, и его выход сбросится в 0. Когда n1 в конце концов поднимется, *Y*

вернется в 1. Как показано на временных диаграммах на **рис. 2.76**, Y начинается с 1 и заканчивается 1, но на короткое время переключается в 0.

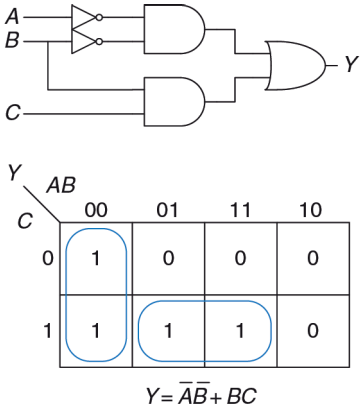


Рис. 2.75 **Схема, подверженная импульсным помехам**

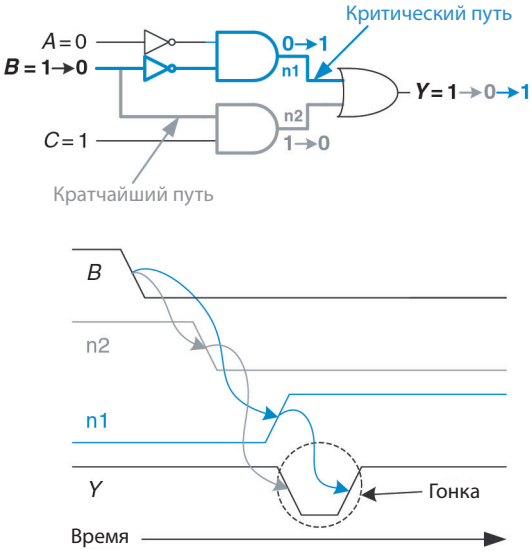


Рис. 2.76 **Временная диаграмма импульсной помехи**

До тех пор, пока мы выдерживаем интервал, равный времени задержки распространения, прежде чем использовать значение с выхода, импульсная помеха не представляет проблемы, потому что выход в конце концов установится в правильное значение.

При желании мы можем избежать этого импульса добавлением дополнительного элемента в схему. Это проще понять с помощью карты Карно.

На **рис. 2.77** показано, как изменение входа B при переходе из $ABC = 001$ в $ABC = 011$ приводит к переходу от одной первичной импликанты к другой. Переход через границу двух первичных импликант в карте Карно свидетельствует о возможном появлении импульсной помехи.

Как показано на временных диаграммах на **рис. 2.76**, если схема реализации одной первичной импликанты выключается до того, как может включиться схема другой первичной импликанты, возникнет импульсная помеха. Чтобы исправить это, мы добавили другую цепь, которая охватывает границу первичных импликант, как показано на **рис. 2.78**. Вы могли бы узнать в этом теорему согласованности, где добавленный терм $\overline{A}C$ – это согласованный или избыточный терм.

На **рис. 2.79** показана схема, устойчивая к паразитным импульсам. Добавленный элемент И выделен синим. Сейчас переключение B, когда $A = 0$ и $C = 1$, не вызывает паразитного импульса на выходе, поскольку синий элемент И формирует на выходе 1 во время этого перехода.

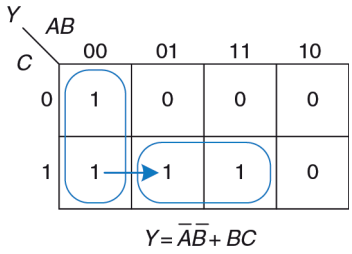


Рис. 2.77 Переход от одной импликанты к другой

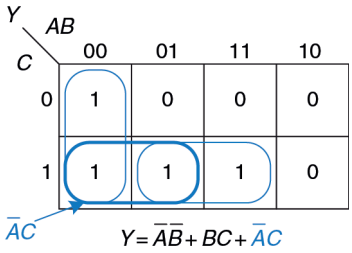


Рис. 2.78 Карта Карно без импульсных помех

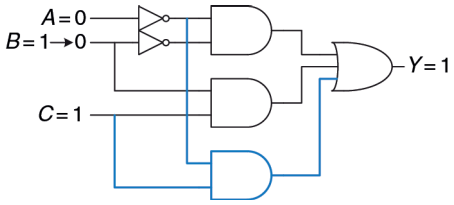


Рис. 2.79 Схема без импульсных помех

В общем случае паразитный импульс может возникать, когда одна переменная пересекает границу между двумя первичными импликантами в карте Карно. Мы можем устранить эти импульсы добавлением избы-

точных импликант в карту Карно, чтобы покрыть эти границы. Естественно, это будет сделано ценой дополнительных аппаратных затрат.

Одновременное переключение нескольких входов также может стать причиной паразитных импульсов. Эти импульсы не могут быть исправлены дополнительными элементами в схеме. Поскольку подавляющее большинство интересующих нас систем имеют одновременные (или почти одновременные) переключения множества входов, возникновение паразитных импульсов в них неизбежно. Хотя мы показали, как устранить один вид импульсных помех, смысл дискуссии о паразитных импульсах не в том, чтобы устранять их, а в том, чтобы знать, что они есть. Это особенно важно при анализе временных диаграмм в симуляторе или на экране осциллографа.

2.10. Заключение

Цифровая схема – это модуль с дискретными значениями входов и выходов и спецификацией, описывающей его функциональные и временные характеристики. Эта глава посвящена комбинационным схемам, выходы которых зависят только от значений на их входах в текущий момент.

Функциональное описание комбинационной схемы может быть задано таблицей истинности или логическим выражением. Логическое выражение для любой таблицы истинности может быть получено в виде совершенной дизъюнктивной нормальной формы или совершенной конъюнктивной нормальной формы. В первом случае функция записывается как дизъюнкция конъюнкций, то есть логическая сумма (логическое «ИЛИ») одной или более импликант. Импликанта представляет собой произведение (логическое «И») литералов. Литералы же – это прямая или комплементарная форма входных переменных.

Логические выражения могут быть упрощены, используя правила булевой алгебры. В частности, их можно упростить, объединяя импликанты, которые отличаются только прямой и комплементарной формами одного из литералов: $PA + P\overline{A} = P$.

Карты Карно – визуальный инструмент для минимизации функций от двух до четырех переменных. На практике разработчики обычно могут упростить функции нескольких переменных «в уме», исходя только из своего опыта. Системы автоматизированного проектирования используются для более сложных функций; эти методы и инструменты обсуждаются в **главе 4**.

Логические элементы соединяют для того, чтобы создать комбинационную схему, которая выполняет требуемую логическую функцию. Любая функция в дизъюнктивной нормальной форме может быть построена, используя двухуровневую логику: элемент НЕ образует комплементарную форму входов, элемент И формирует произведения, и элемент ИЛИ

формирует сумму. В зависимости от функции и доступности базовых элементов многоуровневая логическая реализация с элементами разных типов может оказаться более эффективной. Например, для КМОП-схем больше подходят элементы И-НЕ и ИЛИ-НЕ, потому что эти элементы могут быть построены напрямую на КМОП-транзисторах без использования дополнительного инвертора. Когда используются элементы И-НЕ и ИЛИ-НЕ, для сокращения числа инверторов полезно применять перемещение инверсии.

Логические элементы комбинируются, чтобы создать более сложные схемы, такие как мультиплексоры, дешифраторы и схемы приоритета. Мультиплексор выбирает один из входов данных, основываясь на входе управления. Дешифратор устанавливает один из выходов в ВЫСОКОЕ значение в соответствии со входами. Приоритетная схема выдает 1 на выход, указывающий на вход с самым высоким приоритетом. Все эти схемы – примеры комбинационных «строительных блоков». В **главе 5** вы познакомитесь с еще большим количеством «строительных блоков», включая различные арифметические схемы. Эти блоки будут широко использоваться при создании микропроцессора в **главе 7**.

Временные характеристики комбинационной схемы включают в себя задержки распространения и отклика. Они указывают на наибольшее и наименьшее время между изменением входа и соответствующим изменением выходов. Вычисление задержки распространения заключается в определении критического пути в схеме и затем в сложении вместе задержек всех элементов на этом пути. Существует множество различных способов реализации сложной комбинационной схемы; эти способы предполагают достижение компромисса между ее скоростью работы и ценой.

В следующей главе будут рассмотрены последовательностные схемы, чьи выходы зависят как от текущих значений входов, так и от всей предыстории (последовательности) изменений сигналов на входах. Другими словами, мы рассмотрим схемы, обладающие свойством памяти.

Упражнения

Упражнение 2.1 Запишите логическое выражение в совершенной дизъюнктивной нормальной форме для всех таблиц истинности, приведенных на **рис. 2.80**.

Упражнение 2.2 Запишите логическое выражение в совершенной дизъюнктивной нормальной форме для всех таблиц истинности, приведенных на **рис. 2.81**.

Упражнение 2.3 Запишите логическое выражение в совершенной конъюнктивной нормальной форме для всех таблиц истинности, приведенных на **рис. 2.80**.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рис. 2.80 Таблицы истинности для упражнений 2.1 и 2.3

Упражнение 2.4 Запишите логическое выражение в совершенной конъюнктивной нормальной форме для всех таблиц истинности, приведенных на **рис. 2.81**.

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Рис. 2.81 Таблицы истинности для упражнений 2.2 и 2.4

Упражнение 2.5 Минимизируйте все логические выражения, полученные в **упражнении 2.1**.

Упражнение 2.6 Минимизируйте все логические выражения, полученные в **упражнении 2.2**.

Упражнение 2.7 Нарисуйте простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.5**. Под простой схемой подразумевается такая, которая состоит из небольшого количества элементов, но при этом ее разработчик не тратит много времени на проверку каждой из возможных реализаций схемы.

Упражнение 2.8 Нарисуйте комбинационные схемы, реализующие выражения, полученные в **упражнении 2.6**.

Упражнение 2.9 Повторите **упражнение 2.7**, используя только элементы НЕ, И и ИЛИ.

Упражнение 2.10 Повторите **упражнение 2.8**, используя только элементы НЕ, И и ИЛИ.

Упражнение 2.11 Повторите **упражнение 2.7**, используя только элементы НЕ, И-НЕ и ИЛИ.

Упражнение 2.12 Повторите **упражнение 2.8**, используя только элементы НЕ, И-НЕ и ИЛИ.

Упражнение 2.13 Упростите следующие логические выражения, используя теоремы булевой алгебры. Проверьте правильность результатов, используя таблицы истинности или карты Карно.

a) $Y = AC + \overline{A}\,\overline{B}C$.

b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$.

c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$.

Упражнение 2.14 Упростите следующие логические выражения, используя теоремы булевой алгебры. Проверьте правильность результатов, используя таблицы истинности или карты Карно.

a) $Y = \overline{A}BC + \overline{A}B\overline{C}$.

b) $Y = \overline{ABC} + A\overline{B}$.

c) $Y = ABC\overline{D} + \overline{ABCD} + \overline{(A + B + C + D)}$.

Упражнение 2.15 Нарисуйте простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.13**.

Упражнение 2.16 Нарисуйте простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.14**.

Упражнение 2.17 Упростите каждое из следующих логических выражений. Нарисуйте простые комбинационные схемы, реализующие полученные выражения.

a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$.

b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$.

c) $Y = ABC + ABD + ABE + ACD + ACE + \overline{(A + D + E)} + \overline{B}\,\overline{C}D + \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{B}\,\overline{D}\,\overline{E}$.

Упражнение 2.18 Упростите каждое из следующих логических выражений. Нарисуйте простые комбинационные схемы, реализующие полученные выражения.

a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$.

b) $Y = \overline{(A + B + C)}D + AD + \overline{B}$.

c) $Y = ABCD + \overline{A}B\overline{C}D + \overline{(\overline{B} + D)}E$.

Упражнение 2.19 Приведите пример таблицы истинности, содержащей от 3 до 5 млрд строк, которая может быть реализована схемой, использующей менее 40 двухвходовых логических элементов (но не менее одного).

Упражнение 2.20 Приведите пример схемы с циклическим путем, которая при этом является комбинационной.

Упражнение 2.21 Алиса Хакер утверждает, что любое логическое выражение может быть записано в виде минимальной дизъюнктивной нормальной формы, то есть в виде логической суммы простых импликант. Бен Битдидл утверждает, что существуют такие выражения, минимальные формы которых не содержат все простые импликанты. Объясните, почему Алиса права, или приведите контрпример, подтверждающий точку зрения Бена.

Упражнение 2.22 Докажите следующие теоремы, используя совершенную индукцию. Вам не надо доказывать двойственные им теоремы:

a) теорема об идемпотентности (T3);
b) теорема дистрибутивности (T8);
c) теорема склеивания (T10).

Упражнение 2.23 Докажите теорему де Моргана (T12) для трех переменных, используя совершенную индукцию.

Упражнение 2.24 Напишите логические выражения для схемы, показанной на **рис. 2.82**. Вы не должны минимизировать эти выражения.

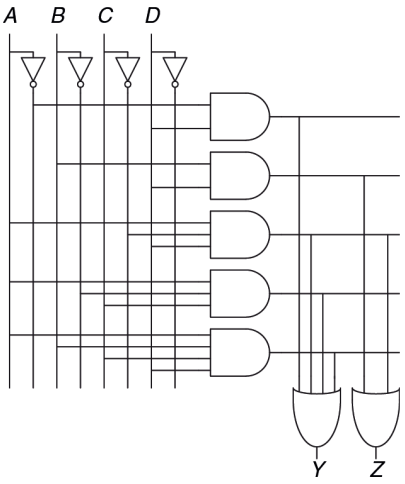


Рис. 2.82 Принципиальная схема

Упражнение 2.25 Минимизируйте логические выражения, полученные в **упражнении 2.24**, и нарисуйте усовершенствованную схему, реализующую эти функции.

Упражнение 2.26 Используя элементы, эквивалентные по де Моргану, и метод перемещения инверсии, перерисуйте схему, приведенную на **рис. 2.83**, чтобы вы могли найти ее логическое выражение «на глаз». Запишите это логическое выражение.

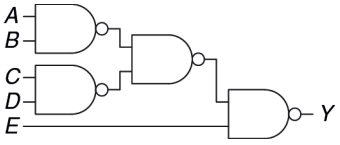


Рис. 2.83 Принципиальная схема

Упражнение 2.27 Повторите **упражнение 2.26** для схемы на **рис. 2.84**.

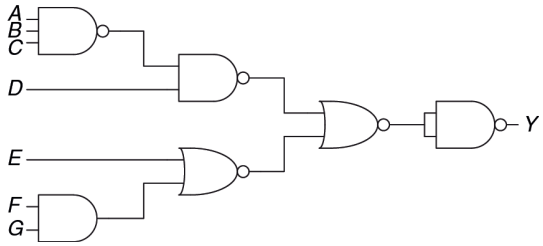


Рис. 2.84 Принципиальная схема

Упражнение 2.28 Найдите минимальное логическое выражение для функции, заданной на **рис. 2.85**. Не забудьте при этом воспользоваться наличием безразличных значений в таблице истинности.

A	*B*	*C*	*D*	*Y*
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рис. 2.85 Таблица истинности

Упражнение 2.29 Нарисуйте схему, реализующую функцию, полученную в **упражнении 2.28**.

Упражнение 2.30 Могут ли в схеме из **упражнения 2.29** появиться потенциальные паразитные импульсы при изменении состояния одного из входов? Если нет, объясните, почему. Если да, покажите, как надо изменить схему, чтобы устранить паразитные импульсы.

Упражнение 2.31 Найдите минимальное логическое выражение для функции, заданной на **рис. 2.86**. Не забудьте при этом воспользоваться наличием безразличных значений в таблице истинности.

Упражнение 2.32 Нарисуйте схему, реализующую функцию, полученную в **упражнении 2.31**.

Упражнение 2.33 Бен Битдидл будет наслаждаться пикником в солнечный день, если не будет муравьев. Он также будет наслаждаться пикником в любой день, если увидит колибри, а еще в те дни, когда есть муравьи и божьи коровки. Запишите логическое выражение для расчета радости (E) в терминах наличия солнца (S), муравьев (A), колибри (H) и божьих коровок (L).

Упражнение 2.34 Завершите разработку дешифратора семисегментного индикатора для сегментов от S_c до S_g (**пример 2.10**).

a) Запишите логическое выражение для выходов от S_c до S_g при условии, что при подаче на вход значения более 9 выход должен быть нулем.
b) Запишите логическое выражение для выходов от S_c до S_g при условии, что при подаче на вход значения более 9 состояние выхода безразлично.
c) Нарисуйте простую реализацию на уровне логических элементов для случая (b). При необходимости используйте общие логические элементы для нескольких выходов.

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рис. 2.86 Таблица истинности

Упражнение 2.35 Схема имеет четыре входа и два выхода. На входы $A_{3:0}$ подается число от 0 до 15. Выход P должен быть равен ИСТИНЕ, если число на входе простое (0 и 1 не являются простыми, а 2, 3, 5 и так далее – являются). Выход D должен быть равен ИСТИНЕ, если число делится на 3. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.36 Приоритетный шифратор имеет 2^N входов. Он формирует на N-разрядном выходе номер самого старшего входного бита, который принимает значение ИСТИНА. Он также формирует на выходе NONE значение ИСТИНА, если ни один из входов не принимает значение ИСТИНА. Разработайте восьмивходовый приоритетный шифратор с входом $A_{7:0}$ и выходами $Y_{2:0}$ и NONE. Например, если вход A принимает значение 00100000, то выход Y должен быть 101, а NONE – 0. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.37 Разработайте модифицированный приоритетный шифратор (**упражнение 2.36**), который имеет 8-разрядный вход $A_{7:0}$, а также 3-разрядные выходы $Y_{2:0}$ и $Z_{2:0}$. На выходе Y формируется номер самого старшего входного бита, который принимает значение ИСТИНА. На выходе Z формируется номер второго по старшинству входного бита, который принимает значение ИСТИНА. Y принимает значение 0, если все биты входа – ЛОЖЬ. Z принимает значение 0, если только один бит входа – ИСТИНА. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.38 M-битный унарный код числа k содержит k единиц в младших разрядах и $(M - k)$ нулей во всех старших разрядах. Преобразователь бинарного кода в унарный имеет N входов и $(2^N - 1)$ выходов. Он формирует $(2^N - 1)$-битный унарный код для числа, установленного на входе. Например, если на входе 110, то на выходе должно быть 0111111. Разработайте преобразователь трехбитного бинарного кода в семибитный унарный. Запишите логическое выражение для каждого из выходов и нарисуйте схему.

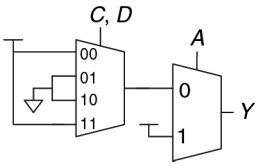


Рис. 2.87 Схема на мультиплексорах

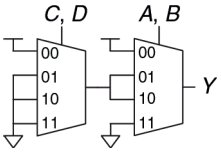


Рис. 2.88 Схема на мультиплексорах

Упражнение 2.39 Запишите минимизированное логическое выражение для функции, выполняемой схемой, показанной на **рис. 2.87**.

Упражнение 2.40 Запишите минимизированное логическое выражение для функции, выполняемой схемой, показанной на **рис. 2.88**.

Упражнение 2.41 Разработайте схему, реализующую функцию, описанную на **рис. 2.80 (b)**, используя:

a) восьмивходовый мультиплексор (8:1);
b) четырехвходовый мультиплексор (4:1) и один инвертор;
c) двухвходовый мультиплексор (2:1) и два любых других логических элемента.

Упражнение 2.42. Разработайте схему, реализующую функцию из **упражнения 2.17 (a)**, используя:

a) восьмивходовый мультиплексор (8:1);
b) четырехвходовый мультиплексор (4:1) без других логических элементов;
c) двухвходовый мультиплексор (2:1), один элемент ИЛИ и один инвертор.

Упражнение 2.43 Рассчитайте задержку распространения t_{pd} и задержку реакции t_{cd} для схемы на **рис. 2.83**. Значения задержек элементов даны в **табл. 2.8**.

Таблица 2.8 Значения задержек элементов для упражнений 2.43–2.47

Элемент	t_{pd} (пс)	t_{cd} (пс)
НЕ	15	10
Двухвходовый И-НЕ	20	15
Трехвходовый И-НЕ	30	25
Двухвходовый ИЛИ-НЕ	30	25
Трехвходовый ИЛИ-НЕ	45	35
Двухвходовый И	30	25
Трехвходовый И	40	30
Двухвходовый ИЛИ	40	30
Трехвходовый ИЛИ	55	45
Двухвходовый Исключающее ИЛИ	60	40

Упражнение 2.44 Рассчитайте задержку распространения и задержку реакции для схемы на **рис. 2.84**. Значения задержек элементов даны в **табл. 2.8**.

Упражнение 2.45 Нарисуйте схему для быстродействующего дешифратора 3:8. Значения задержек элементов даны в **табл. 2.8** (используйте только указанные в таблице элементы). Разработайте дешифратор таким образом, чтобы он имел минимальный возможный критический путь, и найдите этот путь. Каковы задержки распространения и реакции у схемы?

Упражнение 2.46 Измените схему из **упражнения 2.35**, чтобы она была максимально быстродействующей. Используйте только элементы из **табл. 2.8**. Нарисуйте новую схему и определите критический путь. Каковы задержки распространения и реакции у схемы?

Упражнение 2.47 Измените приоритетный дешифратор из **упражнения 2.36**, чтобы он работал максимально быстро. Используйте только элементы из **табл. 2.8**. Нарисуйте новую схему и определите критический путь. Каковы задержки распространения и реакции у схемы?

Упражнение 2.48 Разработайте восьмивходовый мультиплексор так, чтобы задержка от входов до выходов была минимальной. Используйте только элементы из **табл. 2.7**. Нарисуйте схему. Используя значения задержек элементов из таблицы, определите задержку от входов до выходов.

Вопросы для собеседования

Здесь представлены примеры вопросов, которые могут быть заданы соискателям при поиске работы в области разработки цифровых устройств.

Вопрос 2.1 Нарисуйте схему, реализующую функцию «Исключающее ИЛИ», используя логические элементы И-НЕ. Какое минимальное количество элементов И-НЕ для этого требуется?

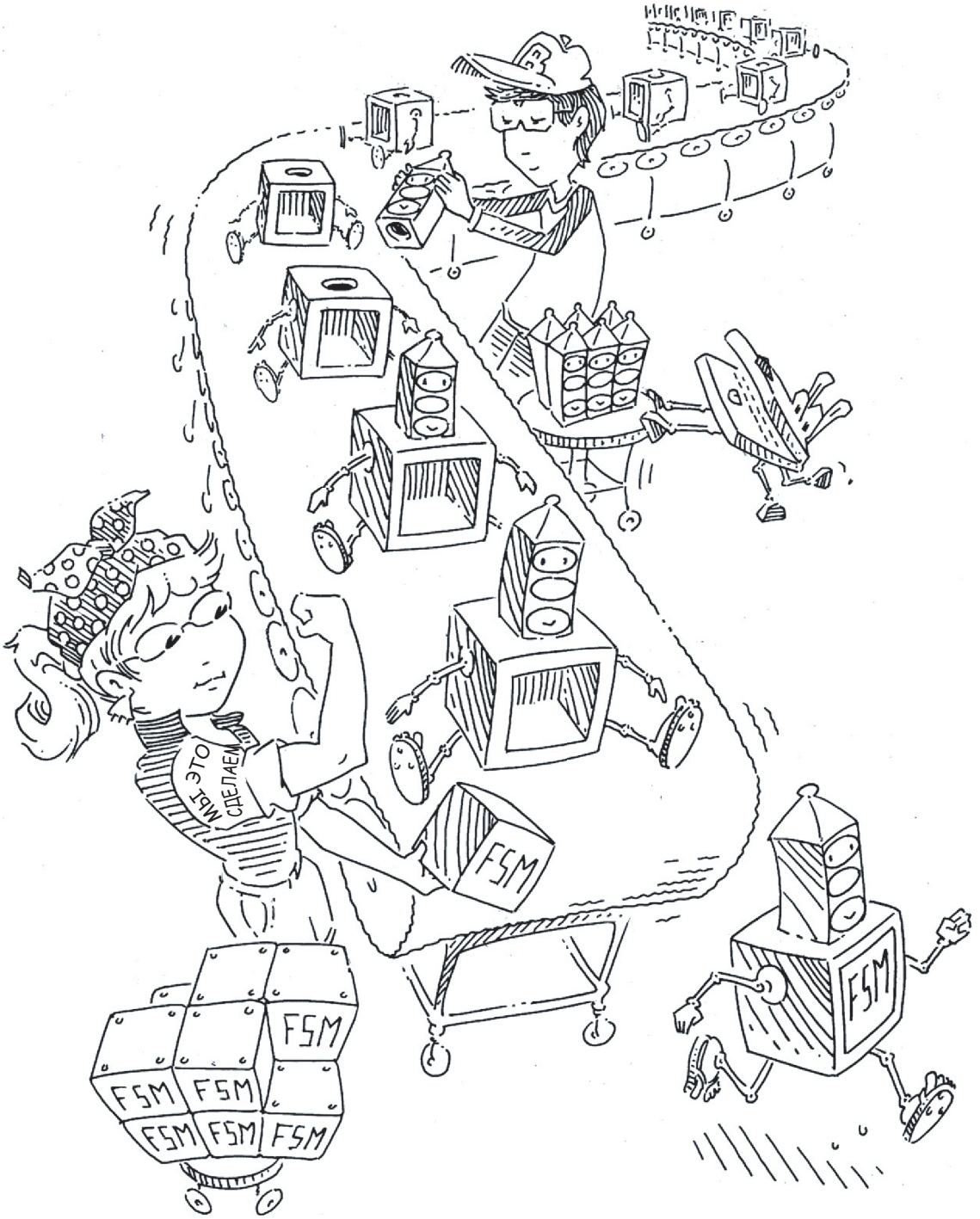
Вопрос 2.2 Разработайте схему, которая показывает, содержит ли заданный месяц 31 день. Месяц задается 4-разрядным входом $A_{3:0}$. Например, значению 0001 на входе соответствует месяц январь, а значению 1100 – декабрь. Выход схемы Y должен принимать значение ИСТИНА только тогда, когда на вход подан номер месяца, в котором 31 день. Опишите логическую функцию, минимизируйте ее, и нарисуйте логическую схему, используя минимальное количество элементов (*подсказка*: не забудьте воспользоваться безразличными состояниями).

Вопрос 2.3 Что такое буфер с тремя состояниями? Как и для чего он используется?

Вопрос 2.4 Элемент или набор элементов является универсальным, если он может быть использован для реализации любой логической функции. Например, набор {И, ИЛИ, НЕ} является универсальным.

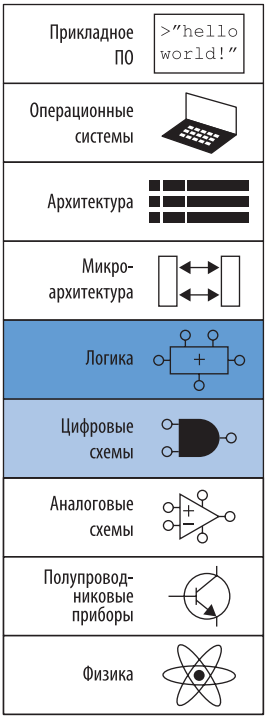
a) Является ли элемент И универсальным? Почему?
b) Является ли набор элементов {ИЛИ, НЕ} универсальным? Почему?
c) Является ли элемент И-НЕ универсальным? Почему?

Вопрос 2.5 Объясните, почему задержка реакции схемы может быть меньше или равна задержке распространения.


МЫ ЭТО СДЕЛАЕМ
FSM
FSM
FSM
FSM
FSM
FSM
FSM
FSM

Глава 3

Разработка последовательностной логики



3.1. Введение

В предыдущей главе мы рассмотрели процесс анализа и разработки комбинационных логических схем. Значение на выходе комбинационной схемы зависит лишь от значений на входе в текущий момент времени. Мы можем создать оптимизированную схему согласно техническому заданию в виде таблицы истинности или в виде логического выражения.

В этой главе мы будем анализировать и разрабатывать *последовательные логические схемы*. Значение на выходе последовательностной логической схемы зависит как от текущих, так и от предыдущих входных значений, следовательно, последовательностные логические схемы обладают памятью. Последовательностные логические схемы могут явно

запоминать предыдущие значения определенных входов, а могут «сжимать» предыдущие значения определенных входов в меньшее количество информации, называемое *состоянием системы*. Состояние цифровой последовательностной схемы – набор битов, называемый *переменными состояния*. Эти биты содержат всю информацию о прошлом, необходимую для определения будущего поведения схемы.

Глава начинается с изучения защелок и триггеров. Они являются простыми последовательностными схемами, запоминающими один бит информации. Вообще говоря, последовательностные схемы достаточно сложно анализировать. С целью упрощения разработки мы ограничимся только синхронными схемами, состоящими из комбинационной логики и набора триггеров, хранящих информацию о состоянии системы. В главе описываются конечные автоматы, с помощью которых можно легко и просто разрабатывать последовательностные схемы. Наконец, мы проанализируем быстродействие последовательностных схем и обсудим параллельные вычисления как способ повышения быстродействия.

3.2. Защелки и триггеры

Основным блоком для построения памяти является бистабильная ячейка – элемент с двумя устойчивыми состояниями. На **рис. 3.1 (a)** показана простая бистабильная ячейка, состоящая из пары инверторов, замкнутых в кольцо. Эту схему можно перерисовать так, чтобы рисунок выглядел симметрично (**рис. 3.1 (b)**). Теперь видно, что инверторы соединены перекрестно, то есть вход I1 соединен с выходом I2 и наоборот. У схемы нет ни одного входа, зато есть два выхода Q и $\overline{Q}$. Анализ этой схемы отличается от анализа комбинационной схемы, так как схема является циклической: Q зависит от $\overline{Q}$, а $\overline{Q}$ зависит от Q.

Выход последовательностной схемы принято обозначать буквой Q аналогично этому, выход комбинационной схемы принято обозначать буквой Y.

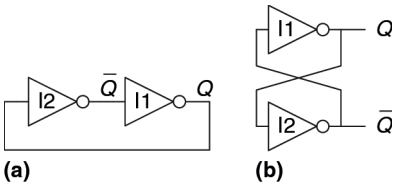


Рис. 3.1 Перекрестно соединенные инверторы

Рассмотрим два случая: $Q = 0$ и $Q = 1$.

- **Случай I:** $Q = 0$
  Как показано на **рис. 3.2 (a)**, на вход I2 поступает сигнал $Q = 0$. I2 инвертирует сигнал и подает на вход I1 сигнал $\overline{Q} = 1$. Соответственно, на выходе I1 – логический 0. В рассмотренном случае схема находится в *стабильном состоянии*.

- **Случай II:** $Q = 1$
  Как показано на **рис. 3.2 (b)**, на вход I2 поступает 1 (Q). I2 инвертирует сигнал и подает на вход I1 – 0 ($\overline{Q}$). Соответственно, на выходе I1 – логическая 1. В этом случае схема также находится в стабильном состоянии.

Так как инверторы, включенные перекрестно, имеют два стабильных состояния $Q = 0$ и $Q = 1$, то говорят, что схема бистабильна. У схемы есть и третье состояние, когда оба выхода находятся в состоянии между 0 и 1. Такое состояние называется *метастабильным*, и оно будет рассмотрено в **разделе 3.5.4**.

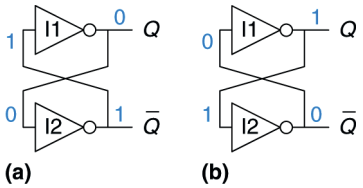


Рис. 3.2 Бистабильный режим перекрестно соединенных инверторов

Элемент с N стабильными состояниями хранит $\log_2 N$ бит информации. Таким образом, бистабильная ячейка хранит 1 бит. Состояние перекрестно включенных инверторов содержится в одной переменной состояния Q. Значение Q сообщает нам всю информацию о прошлом, необходимую для определения будущего поведения схемы. В частности, если $Q = 0$, то оно и будет 0 всегда, а если $Q = 1$, то оно и останется 1. У схемы есть еще один выход – $\overline{Q}$. Но $\overline{Q}$ не содержит никакой дополнительной информации, так как если Q известно, то $\overline{Q}$ определено однозначно. С другой стороны, $\overline{Q}$ можно было бы также рассматривать как переменную состояния.

При включении питания исходное состояние последовательностной схемы неизвестно и обычно непредсказуемо. Оно может быть различным всякий раз, когда схему включают.

Несмотря на то что перекрестно включенные инверторы могут хранить бит информации, они не используются на практике, так как у схемы нет входов, с помощью которых пользователь мог бы контролировать ее состояние. Однако другие элементы, такие как защелки и триггеры, имеют входы, которые позволяют управлять переменной состояния. Эти схемы рассматриваются в оставшейся части раздела.

3.2.1. RS-триггер

Одной из простейших последовательностных схем является *RS-триггер* (от англ. *Reset* и *Set*), состоящий, как показано на **рис. 3.3**, из двух перекрестно включенных элементов ИЛИ-НЕ. У RS-триггера есть два входа R и S и два выхода Q и $\overline{Q}$. Принципы работы RS-триггера и схемы с перекрестно включенными инверторами аналогичны, но со-

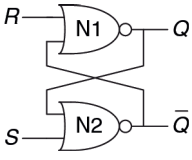


Рис. 3.3 RS-триггер (защелка)

стояние защелки контролируется входами R и S, которые сбрасывают и устанавливают выход Q.

Для того чтобы понять, как работает неизвестная цепь, обычно строят ее таблицу истинности. Вспомним, что на выходе элемента ИЛИ-НЕ появляется логический ноль, если на какой-либо из его входов подана логическая единица. Рассмотрим четыре возможные комбинации R и S.

- **Случай I:** $R = 1, S = 0$
  На входе N1 как минимум одна единица – вход R, следовательно, выход $Q = 0$. Оба входа N2 – в состоянии логического нуля ($Q = 0$ и $S = 0$), поэтому выход $\overline{Q} = 1$.
- **Случай II:** $R = 0, S = 1$
  На вход N1 поступает 0 и $\overline{Q}$. Так как мы еще не знаем значения $\overline{Q}$, мы не можем определить значение Q. На вход N2 поступает как минимум одна единица S, поэтому на выходе $\overline{Q}$ – логический 0. Теперь можно вернуться к определению состояния выхода элемента N1. Мы знаем, что на обоих его входах 0, следовательно, $Q = 1$.
- **Случай III:** $R = 1, S = 1$
  Как на входе N1, так и на входе N2 как минимум по одной единице (R и S), поэтому на выходе каждой защелки – логический 0. Следовательно, $Q = 0$ и $\overline{Q} = 0$.
- **Случай IV:** $R = 0, S = 0$
  На вход N1 поступает 0 и $\overline{Q}$. Так как мы еще не знаем значения $\overline{Q}$, мы не можем определить значение на выходе элемента N1. На вход N2 поступает 0 и Q. Так как мы еще не знаем значения Q, мы не можем определить значение на выходе элемента N2. Кажется, мы зашли в тупик. Этот случай аналогичен случаю с двумя перекрестно включенными инверторами. Мы знаем, что Q должен быть равен либо 0, либо 1. Итак, мы сможем решить проблему, если рассмотрим каждый из этих двух случаев отдельно.
  - **Случай IV (a):** $Q = 0$
    Так как S и Q равны 0, то на выходе N2 будет логическая 1, $\overline{Q} = 1$, как показано на **рис. 3.4 (a)**. Теперь на входе N1 есть одна единица – $\overline{Q}$, поэтому на его выходе $Q = 0$, как мы и предполагали.
  - **Случай IV (b):** $Q = 1$
    Так как $Q = 1$, то на выходе N2 будет 0, $\overline{Q} = 0$, как показано на **рис. 3.4 (b)**. Теперь на обоих входах N1 нули (R и $\overline{Q}$), поэтому на его выходе логическая 1, $Q = 1$, как мы и предполагали.

  Исходя из сказанного выше, предположим, что у Q есть какое-то определенное значение, установленное до наступления случая IV, которое мы назовем $Q_{пред}$. $Q_{пред}$ может быть либо 0, либо 1. $Q_{пред}$ отражает состояние системы. Когда R и S равны 0, на выходе Q будет сохраняться старое значение $Q_{пред}$, а $\overline{Q}$ будет его противоположным значением.

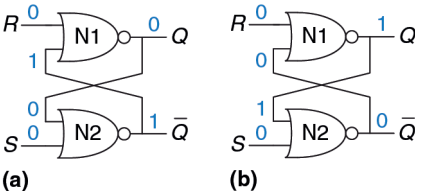


Рис. 3.4 Бистабильные состояния RS-триггера

Таблица истинности, приведенная на **рис. 3.5**, иллюстрирует эти четыре случая. Входы R и S отвечают за сброс и установку значений соответственно.

Установить бит означает перевести его в логическую единицу, а сбросить – в логический ноль. Обычно $\overline{Q}$ является булевым дополнением Q. Когда поступает команда сброса $R = 1$, выход Q принимает значение 0, а выход $\overline{Q}$ – противоположное (логическую 1). Когда поступает команда установки бита $S = 1$, выход Q становится единицей, а $\overline{Q}$ – нулем. Если ни на один из входов не поступает логическая единица, на обоих выходах сохраняется предыдущее значение $Q_{пред}$. Подача на входы одновременно $R = 1$ и $S = 1$ не имеет особого смысла, так как это означает, что выход должен быть одновременно и установлен, и сброшен, что невозможно. Защелка, не зная, что ей делать, выставляет как на прямом, так и на инверсном выходе логический 0.

Условное обозначение RS-триггера представлено на **рис. 3.6**. Условные обозначения используются при модульном проектировании схемы с целью абстрагирования от внутренней структуры элемента.

Случай	S	R	Q	$\overline{Q}$
IV	0	0	$Q_{пред}$	$\overline{Q}_{пред}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Рис. 3.5 Таблица истинности RS-триггера

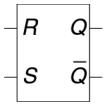


Рис. 3.6 Обозначение RS-триггера

Существует несколько способов построения RS-триггера, таких как использование логических элементов или транзисторов. Тем не менее любой элемент схемы, специфицированный таблицей истинности на **рис. 3.5**, обозначается символом на **рис. 3.6** и называется RS-триггером.

Так же как и перекрестно включенные инверторы, RS-триггер является бистабильным элементом с одним битом состояния, хранящимся в Q. Состоянием можно управлять при помощи входов R и S. Когда на R поступает высокий уровень сигнала, выход сбрасывается в 0. Когда высокий уровень сигнала приходит на S, выход устанавливается в 1. Если ни на один вход не пришла логическая единица, триггер сохраняет свое предыдущее состояние, значение выходов не изменяется. Отметим, что вся история сигналов, поданных на вход, определяется в одной перемен-

ной состояния Q. Не имеет значения, что происходило в прошлом. Все, что нужно, чтобы предсказать будущее поведение RS-триггера, – это знать, было ли последнее изменение состояния триггера сбросом или установкой.

3.2.2. D-защелка

RS-триггер неудобен из-за необычного поведения, если на оба входа триггера одновременно поступает высокий уровень сигнала. Более серьезная проблема состоит в том, что понятия *ЧТО* и *КОГДА* в контексте изменения состояния триггера объединены его входами R и S. Подача логической единицы на эти входы определяет не только, *ЧТО* произойдет, но и *КОГДА* это произойдет. Разработка схем упрощается, если сущности *ЧТО* и *КОГДА* разделены. D-триггер-защелка (**рис. 3.7 (a)**) решает эти проблемы. У триггера есть два входа: вход *данных* D, определяющий, каким будет следующее состояние, и вход *тактового сигнала* CLK, определяющий, когда оно изменится.

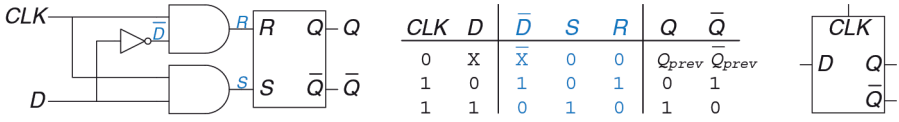


Рис. 3.7 D-триггер-защелка: (a) схема, (b) таблица истинности, (c) обозначение

Для анализа защелки снова составим таблицу истинности (**рис. 3.7 (b)**). Сначала рассмотрим внутренние линии $\overline{D}$, R и S. Если $CLK = 0$, то оба сигнала R и S нулевые, независимо от значения D. Если $CLK = 1$, на выходе одного элемента И будет единица, а на выходе другого – ноль. Элемент И, на выходе которого будет 1, определяется входом D. Значения Q и $\overline{Q}$ определяются R и S по таблице на **рис. 3.5**. Заметим, что пока $CLK = 0$, Q сохраняет предыдущее значение $Q_{пред}$. Если $CLK = 1$, $Q = D$. Очевидно, что $\overline{Q}$ всегда является инверсией Q. В D-защелке исключен случай необычного поведения при одновременно поданных сигналах сброса и установки ($R = 1$ и $S = 1$).

Таким образом, мы видим, что тактовый сигнал контролирует, КОГДА данные проходят через триггер-защелку. Когда $CLK = 1$, защелка «*прозрачна*», т. е. она пропускает данные D на выход Q, как если бы он являлся обычным буфером. Когда $CLK = 0$, защелка «*непрозрачна*», она не пропускает новые данные с входа D на выход Q, а Q сохраняет свое значение. D-защелку иногда называют *прозрачным триггером*, или *триггером с синхронизируемым уровнем*. Условное обозначение D-защелки представлено на **рис. 3.7 (c)**.

Состояние D-триггера-защелки изменяется непрерывно, пока $CLK = 1$. Позже в этой главе мы увидим, что зачастую удобнее изменять состояние схемы только в определенный момент времени. Следующий

раздел – как раз об этом. В нем описывается D-триггер, синхронизируемый фронтом.

Иногда состояние защелки называют «открытым» или «закрытым», а не «прозрачным» или «непрозрачным».

3.2.3. D-триггер

D-триггер, триггер, синхронизируемый фронтом тактового сигнала (далее – триггер), может быть построен из двух включенных последовательно D-защелок. Как показано на **рис. 3.8 (a)**, тактовые сигналы, которые подаются на них, являются инверсией друг друга. Первую защелку называют *ведущей* (master), а вторую – *ведомой* (slave). Защелки соединены линией N1. Условное обозначение D-триггера приведено на **рис. 3.8 (b)**. Когда выход $\overline{Q}$ не используется, обозначение может быть упрощено до представленного на **рис. 3.8 (c)**.

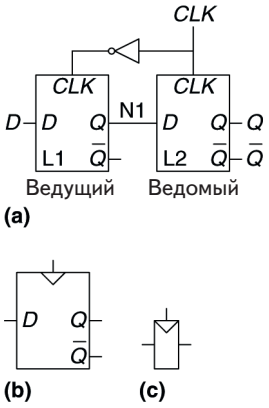


Рис. 3.8 D-триггер: (a) схема, (b) обозначение, (c) упрощенное обозначение

Когда $CLK = 0$, master-защелка открыта, а slave – закрыта. Следовательно, значение с входа D проходит до линии N1. Когда $CLK = 1$, master-защелка закрывается, а slave-защелка открывается. Значение с N1 проходит на выход Q, а N1 при этом становится отрезанным от D. Следовательно, то значение, которое было на входе D непосредственно перед переходом CLK из 0 в 1, сразу же попадает на выход Q, после того как тактовый сигнал устанавливается в 1. Во все остальное время Q сохраняет свое прежнее значение, так как закрытый триггер постоянно блокирует путь между D и Q.

Другими словами, *D-триггер копирует значение с D на Q по переднему фронту тактового импульса и помнит это состояние все остальное время*. Перечитайте это определение несколько раз до тех пор, пока вы его не запомните. Одна из самых распространенных ошибок начинающих разработчиков цифровых схем – они забывают, что такое синхронизация фронтом. Вход D определяет новое, будущее состояние триггера. Передний фронт определяет момент времени, когда состояние будет обновлено.

Различие между триггером и защелкой весьма расплывчатое, оно изменялось с течением времени. В производственных кругах под триггером обычно понимают триггер, синхронизируемый передним фронтом, или, другими словами, это бистабильный элемент с тактовым входом. Состояние триггера изменяется только по переднему фронту тактового сигнала, то есть когда тактовый сигнал переходит из 0 в 1. Бистабильные элементы, в которых отсутствует синхронизация по переднему фронту, обычно называют защелками. Употребляя термины «триггер» или «защелка», обычно имеют в виду D-триггер или D-защелку соответственно, потому что именно эти триггеры чаще всего используются на практике.

D-триггер также известен как *MS-триггер*, *master-slave-триггер* и как *триггер, синхронизируемый передним фронтом*. Треугольник в обозначении указывает на то, что вход синхронизируется передним фронтом. У многих триггеров выход $\overline{Q}$ отсутствует, и их обычно используют, когда $\overline{Q}$ не нужен.

Пример 3.1 КОЛИЧЕСТВО ТРАНЗИСТОРОВ В ТРИГГЕРЕ

Сколько транзисторов содержится в D-триггере, описанном в этой главе?

Решение В элементе ИЛИ-НЕ или И-НЕ используется по 4 транзистора. В инверторе используются два транзистора. Элемент И состоит из элементов И-НЕ и НЕ (инвертора), поэтому в нем используется 6 транзисторов. В RS-защелке – два элемента ИЛИ-НЕ или 8 транзисторов. В D-защелке используется RS-защелка, 2 элемента И и один элемент НЕ, или 22 транзистора. В D-триггере используются две D-защелки и один элемент НЕ, или 46 транзисторов. В **разделе 3.2.7** описываются более эффективные способы реализации триггера на основе КМОП-технологии с использованием проходных ключей.

3.2.4. Регистр

N-разрядный регистр – набор из *N* триггеров с общим тактовым сигналом. Таким образом, все биты регистра обновляются одновременно. Регистр является ключевым блоком при построении большинства последовательностных схем. На **рис. 3.9** показана схема и обозначение 4-разрядного регистра с входами $D_{3:0}$ и выходами $Q_{3:0}$. $D_{3:0}$ и $Q_{3:0}$ являются 4-разрядными шинами.

3.2.5. Триггер с функцией разрешения

У некоторых триггеров имеется еще один вход, называемый *EN*, или *ENABLE* (разрешить). Этот вход определяет, будут данные загружены по переднему фронту тактового сигнала или нет. Когда на *EN* подается логическая единица, то такой D-триггер ведет себя так же, как и обычный D-триггер. Если же на *EN* поступает логический ноль, то триггер игнорирует тактовый сигнал и сохраняет свое состояние. Такие триггеры полезны, если мы хотим загружать значения в триггер только на протяжении какого-то времени, а не по каждому переднему фронту тактового импульса.

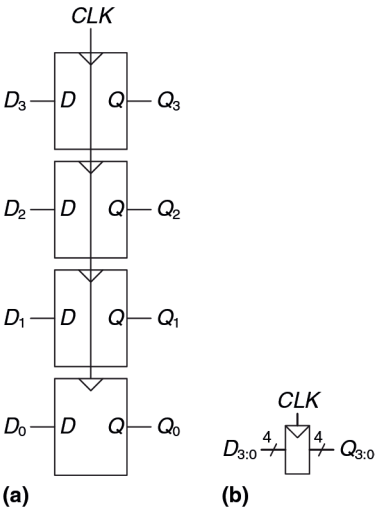


Рис. 3.9 **4-разрядный регистр: (a) схема, (b) обозначение**

На **рис. 3.10** показаны два способа добавления входа разрешения к обычному D-триггеру. На **рис. 3.10 (a)** входной мультиплексор выбирает, подавать ли данные на вход D, если на EN логическая единица, или подавать на вход D старое значение с выхода Q, если на EN подается логический ноль. На **рис. 3.10 (b)** тактовый сигнал проходит, если EN равен единице; импульсы на вход тактового сигнала подаются в обычном режиме. Если на EN – логический ноль, то и на CLK – так же ноль, и триггер сохраняет свое предыдущее состояние. Заметим, что сигнал EN не должен изменяться, пока $CLK = 1$, во избежание сбоя (выброса) тактового сигнала (переключения в неверное время). Вообще говоря, добавление логических элементов в тракт тактового сигнала – плохая идея. Управление тактированием вносит задержку в тактовый сигнал и может привести к временны́м ошибкам, о чем будет сказано в **разделе 3.5.3**, то есть использовать такой метод управления разрешения работы триггера можно только в том случае, если вы понимаете, что делаете. Обозначение триггера с функцией разрешения представлено на **рис. 3.10 (c)**.

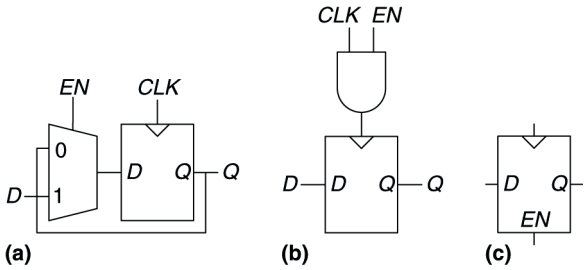


Рис. 3.10 **Триггер с функцией разрешения: (a, b) схемы, (c) обозначение**

3.2.6. Триггер с функцией сброса

В *триггере с функцией сброса* добавляется еще один вход, называемый *RESET* (Сброс). Когда на *RESET* подан 0, сбрасываемый триггер ведет себя как обычный D-триггер. Когда на *RESET* подана 1, такой триггер игнорирует вход D и устанавливает выход в 0. Триггеры с функцией сброса полезны, когда мы хотим ускорить установку определенного состояния (т. е. 0) во всех триггерах системы при первом включении схемы.

Такие триггеры могут сбрасываться как синхронно, так и асинхронно. Синхронно сбрасываемые триггеры сбрасываются только по переднему фронту сигнала CLK. Асинхронно сбрасываемые триггеры сбрасываются сразу же при поступлении логической единицы на вход *RESET*, независимо от тактового сигнала.

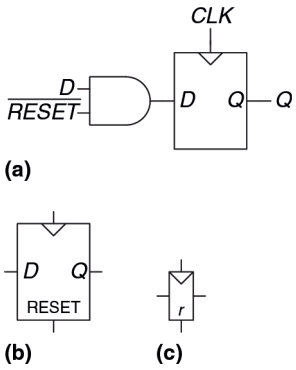


Рис. 3.11 Синхронно сбрасываемый триггер: (a) схема, (b, c) обозначения

На **рис. 3.11 (a)** показано, как построить триггер с синхронным сбросом из обычного D-триггера и элемента И. Когда на $\overline{RESET}$ поступает логический ноль, элемент И подает 0 на вход триггера. Когда на $\overline{RESET}$ поступает логическая единица, элемент И пропускает сигнал *D* на вход триггера. В этом примере $\overline{RESET}$ – сигнал с активным низким уровнем сигнала (инверсная логика). Это означает, что сброс происходит, когда на этот вход поступает 0, а не 1. Добавив инвертор, мы могли бы получить схему с активным высоким уровнем сигнала (прямая логика). На **рис. 3.11 (b)** и **3.11 (c)** показаны обозначения сбрасываемого триггера с прямым сбросом.

Асинхронно сбрасываемые триггеры требуют изменения своей внутренней структуры и оставлены для самостоятельного разбора (**упражнение 3.13**). Они зачастую доступны разработчикам при проектировании цифровых схем как стандартный компонент.

Как вы могли бы легко догадаться, иногда используются и триггеры с функцией установки. Когда установлен сигнал *SET*, в такой триггер загружается логическая 1. Они тоже бывают в синхронном и асинхронном исполнениях. У сбрасываемых и устанавливаемых триггеров также мо-

жет быть вход *ENABLE*, и они могут быть сгруппированы в *N*-разрядные регистры.

3.2.7. Разработка триггеров и защелок на транзисторном уровне

В **примере 3.1** было показано, что если триггеры построены из логических элементов, то в них используется большое количество транзисторов. Но фундаментальная функция защелки (триггера, синхронизируемого уровнем) – быть открытой или закрытой – делает ее схожей с ключом. В **разделе 1.7.7** было указано, что использование проходного логического элемента – эффективный способ создать КМОП-ключ. Следовательно, мы можем воспользоваться преимуществами проходных ключей с целью уменьшения количества транзисторов.

Как показано на **рис. 3.12 (a)**, компактная D-защелка может быть разработана с использованием одного проходного ключа. Когда $CLK = 1$, а $\overline{CLK} = 0$, проходной ключ замкнут, таким образом, D проходит на Q, и защелка открыта. Когда $CLK = 0$, а $\overline{CLK} = 1$, проходной ключ разомкнут, следовательно, выход Q изолирован от входа D, и защелка закрыта. Однако такой триггер имеет следующие существенные недостатки:

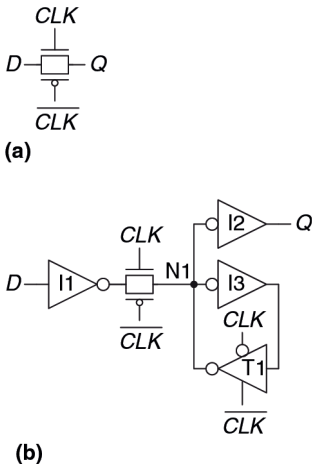


Рис. 3.12 Схема D-триггера-защелки

- **плавающий потенциал на выходе:** когда защелка закрыта, значение выхода Q не подтянуто ни к одному логическому уровню. В этом случае узел Q называют *плавающим*, или *динамическим*. Спустя некоторое время шумы и утечка заряда могут изменить значение выхода Q;
- **отсутствие буферов:** отсутствие буферов приводило к некорректной работе некоторых коммерческих микросхем. Случайный выброс, приводящий к появлению на входе D отрицательного напряжения, может включить n-канальный транзистор, открывая защелку, даже если $CLK = 0$. Аналогично выброс на входе D выше напряжения питания может открыть p-канальный транзистор, даже если $CLK = 0$. Но проходной ключ симметричен, таким образом, он может быть открыт выбросами на выходе Q, тем самым влияя на значения входа D. Основное правило – ни вход проходного ключа, ни узел состояния последовательностной логической схемы никогда не должны применяться там, где существует вероятность возникновения помех или шумов.

На **рис. 3.12 (b)** изображена более надежная 12-транзисторная D-защелка, используемая в современных коммерческих микросхемах. Хотя она и построена на основе тактируемых проходных ключей, в ней добавлены инвертеры I1 и I2, выполняющие роль входного и выходного бу-

На вход этой схемы поступают оба сигнала: CLK и $\overline{CLK}$. Если $\overline{CLK}$ отсутствует, то ставят инвертор, добавляя тем самым еще два транзистора.

феров. Состояние защелки определяется состоянием узла N1. Инвертер I3 и буфер с тремя состояниями T1 образуют обратную связь, тем самым устраняя эффект плавающего потенциала на N1. Если узел N1 отклонится от стационарного состояния под влиянием помех или шума, то, когда CLK будет равен 0, буфер T1 вернет его в это состояние.

На **рис. 3.13** изображен D-триггер, состоящий из двух защелок, управляемых сигналами CLK и $\overline{CLK}$. Мы удалили некоторые лишние инверторы, и теперь для создания триггера требуется лишь 20 транзисторов.

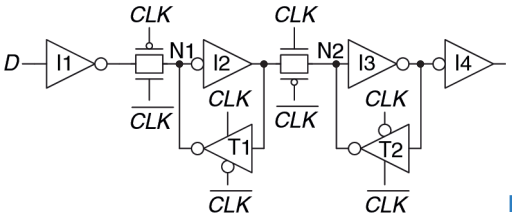


Рис. 3.13 Схема D-триггера

3.2.8. Сравнение защелок и триггеров

Защелки и триггеры являются фундаментальными функциональными узлами последовательностных логических схем. D-защелка открыта, когда $CLK = 1$, тем самым позволяя значению со входа D попасть на выход Q. D-триггер передает значение с D на Q только по переднему фронту тактового сигнала. Во всех остальных случаях триггеры и защелки сохраняют свое предыдущее состояние. Регистром называется набор из нескольких D-триггеров с общим тактовым сигналом.

Пример 3.2 СРАВНЕНИЕ ЗАЩЕЛОК И ТРИГГЕРОВ

Бен Битдидл подал сигналы D и CLK, которые показаны на **рис. 3.14**, на D-защелку и D-триггер. Помогите ему определить значение выхода Q для каждого устройства.

Решение На **рис. 3.15** показаны временные диаграммы выходных сигналов с учетом небольших задержек в триггере и защелке. Стрелки указывают на причину, вызвавшую переключение сигнала на выходе. Исходное значение Q неизвестно, это показано двумя горизонтальными линиями в начале диаграммы. Сначала рассмотрим защелку. Во время прохождения первого фронта тактового сигнала CLK значение $D = 0$, поэтому Q установится в 0. Каждый раз, когда D будет изменяться в то время как $CLK = 1$, Q также будет изменяться. Если D будет изменяться когда $CLK = 0$, изменений на выходе Q не будет. Теперь рассмотрим триггер, синхронизируемый фронтом. Значение на выходе Q становится равным значению на входе D по каждому фронту тактового сигнала CLK. Во всех других случаях Q не изменяется.

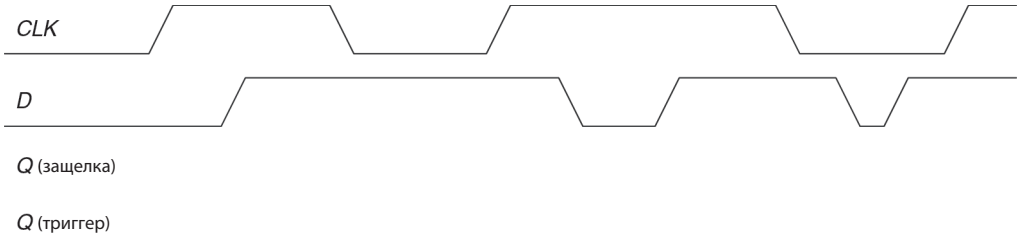


Рис. 3.14 **Исходные временные диаграммы**

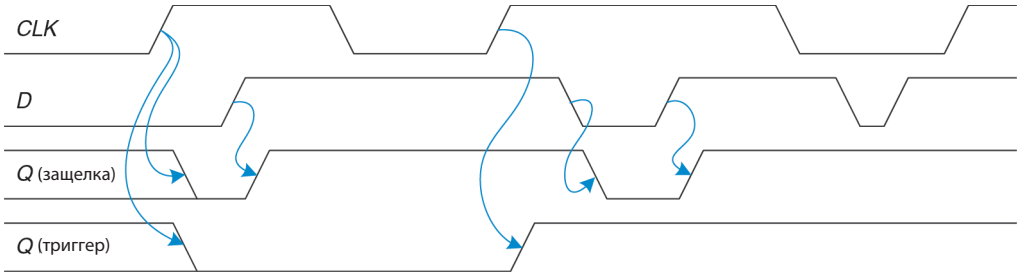


Рис. 3.15 **Решение примера 3.2**

3.3. Разработка синхронных логических схем

Вообще говоря, последовательностные схемы включают в себя все схемы, которые не являются комбинационными, то есть последовательностные схемы – это те, значение выходов которых нельзя однозначно определить, зная лишь текущие значения входов. Поведение некоторых последовательностных схем может быть весьма сложным. Этот раздел начнется с разбора нескольких таких схем. Затем мы введем понятия синхронных последовательностных схем и динамической дисциплины. Ограничив себя рассмотрением только синхронных последовательностных схем, мы сможем сформулировать простые систематические подходы к анализу и разработке таких схем.

3.3.1. Некоторые проблемные схемы

Пример 3.3 НЕУСТОЙЧИВЫЕ СХЕМЫ

Алиса Хакер столкнулась со схемой, которая состоит из трех инверторов, замкнутых в кольцо, как показано на **рис. 3.16**. Выход третьего инвертора подается на вход первого. Задержка распространения каждого из инверторов равна 1 нс. Определите, что происходит в такой схеме.

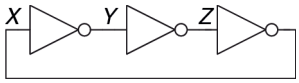


Рис. 3.16 Кольцо из трех инверторов

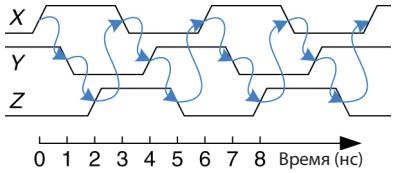


Рис. 3.17 Временные диаграммы кольцевого генератора

Решение Предположим, что в начальный момент времени сигнал X равен логическому 0. Тогда $Y = 1$, $Z = 0$, следовательно, $X = 1$, что расходится с нашим предположением. У этой схемы нет стабильных состояний, поэтому такая схема называется нестабильной, или неустойчивой. На **рис. 3.17** показано поведение схемы. Если сигнал X переходит из 0 в 1 в начальный момент времени, то Y перейдет из 1 в 0 в момент времени $t = 1$ нс, а Z из 0 в 1 – в $t = 2$ нс, а затем X перейдет обратно из 1 в 0 в момент времени $t = 3$ нс. В свою очередь, Y перейдет из 0 в 1 в момент $t = 4$ нс, Z перейдет из 1 в 0 во время $t = 5$ нс, а X снова перейдет из 0 в 1 в момент времени $t = 6$ нс, и далее такое поведение схемы будет повторяться. Каждый узел будет колебаться между 0 и 1 с периодом $T = 6$ нс. Такая схема называется кольцевым генератором.

Период колебаний кольцевого генератора зависит от задержки распространения каждого инвертора. Эта задержка зависит от того, как изготовлен инвертор, от напряжения питания и даже от температуры. Поэтому точно определить период колебаний кольцевого генератора сложно. Иными словами, кольцевой генератор – последовательностная схема без входов и с одним выходом, значения которого периодически изменяются.

Пример 3.4 ГОНКИ В ПОСЛЕДОВАТЕЛЬНОСТНЫХ СХЕМАХ

Бен Битдидл разработал новую D-защелку, которая, как он считает, работает лучше, чем изображенная на **рис. 3.7**, так как в ней используется меньше элементов. Он составил таблицу истинности для выхода Q по данным двух входов D и CLK и предыдущего состояния Q_{prev}. Основываясь на этой таблице, он составил логические уравнения. Для получения Q_{prev} используется обратная связь с выхода Q. Его схема изображена на **рис. 3.18**. Работает ли его защелка корректно, независимо от задержек каждого элемента?

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

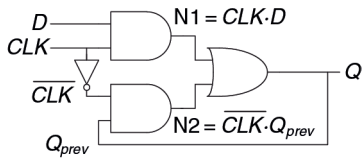


Рис. 3.18 Усовершенствованная D-защелка

Решение На **рис. 3.19** показано, что схема может работать некорректно из-за появления *гонок* (race condition), что приводит к сбою в случае, если определенные элементы медленнее других. Пусть $CLK = D = 1$. Защелка открыта, пропускает данные, и на выходе Q появляется логическая 1. Теперь сигнал CLK переходит из 1 в 0. Триггеру нужно запомнить свое предыдущее значение, т. е. сохранить $Q = 1$. Предположим, что задержка распространения инвертора существенно больше задержек элементов И и ИЛИ. В таком случае сигналы N1 и Q перейдут из 1 в 0 раньше, чем сигнал $\overline{CLK}$ станет 1. В этом случае сигнал N2 никогда не примет значение логической единицы, и выходной сигнал схемы Q останется нулевым.

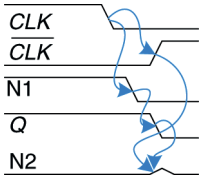


Рис. 3.19 Временные диаграммы защелки, иллюстрирующие гонки

Это пример проекта асинхронной схемы, в которой выходы напрямую связаны обратной связью с входами. Асинхронные схемы не пользуются популярностью из-за непредсказуемости поведения, связанной с быстродействием элементов, когда поведение схемы зависит от того, какой сигнал внутри схемы пройдет быстрее других. Одна схема может работать, при этом другая, кажущаяся идентичной, собранная из элементов с незначительно отличающимися задержками, может не работать. Или схема может работать только при определенных температурах либо напряжениях, при которых задержки соответствуют расчетным. Подобные ошибки проектирования весьма сложно выявлять.

3.3.2. Синхронные последовательностные схемы

В предыдущих двух примерах присутствовали циклические пути, в которых выходы были напрямую соединены обратной связью со входами. Это скорее последовательностные, чем комбинационные схемы. В комбинационной логике нет циклических путей и нет зависимостей состояния выхода от времени прохождения сигнала. Если на входы комбинационной логической схемы поданы определенные сигналы, то ее выходы спустя некоторое время всегда установятся в определенное корректное состояние. Но в последовательностных схемах с циклическими путями может появиться нежелательная нестабильность или гонки. Проверка таких схем требует много времени, и многие выдающиеся разработчики делали подобные ошибки.

Во избежание подобных проблем разработчики разрывают циклические пути и добавляют в разрыв регистры. Это превращает схему в набор комбинационной логики и регистров. В регистрах содержится состояние схемы, изменяющееся только по переднему фронту тактового импульса. В этом случае говорят, что состояние *синхронизировано* с тактовым сигналом. Если период тактового сигнала достаточно большой, чтобы все входы регистров успели установиться до переднего фронта следующего тактового импульса, то эффекты, связанные с гонками, устраняются. Следование правилу «всегда использовать регистры в обратной связи» приводит нас к формальному определению синхронной последовательностной схемы.

Напомним, что схема (цепь) определяется набором входов и выходов, а также функциональными и временными параметрами. У последователь-

ностной схемы существует *конечный набор дискретных состояний* $\{S_0, S_1, ..., S_{k-1}\}$. У *синхронной последовательностной схемы* есть вход тактового сигнала, передние фронты тактовых импульсов определяют последовательность точек на временной оси, в которых происходят изменения состояния. Мы часто будем использовать термины «*текущее состояние*» и «*следующее состояние*», для того чтобы отличать состояние системы в настоящем времени от состояния системы, в которое она перейдет по переднему фронту следующего тактового импульса. Функциональное описание определяет следующее состояние и значение каждого выхода для каждой возможной комбинации текущих состояний и входных сигналов. Временная спецификация состоит из верхней границы t_{pcq} и нижней границы t_{ccq} длительности временного промежутка от переднего фронта тактового импульса до момента изменения *выходного* сигнала, а также из времен *предустановки* и *удержания* t_{setup} и t_{hold}, которые определяют промежуток времени до и после поступления переднего фронта тактового импульса, в течение которого значения на входах не должны изменяться.

t_{pcq} — это задержка распространения тракта «вход тактового сигнала»—«выход *Q*» (до полной установки нового значения) последовательной логической схемы. t_{ccq} — это задержка реакции тракта «вход тактового сигнала»—«выход *Q*». Эти задержки аналогичны задержкам t_{pd} и t_{cd} в комбинационной логике.

Такое определение синхронной последовательностной схемы является достаточным, но в то же время слишком строгим. Например, в высокопроизводительных микропроцессорах некоторые регистры могут получать тактовый сигнал с задержкой. Тактовый сигнал также может подаваться через проходной ключ. Это позволяет добиться максимально возможного быстродействия системы. Также в некоторых микропроцессорах вместо регистров используются защелки. Тем не менее это определение подходит ко всем синхронным последовательностным схемам, рассматриваемым в этой книге, и к большинству коммерческих систем.

Правила *построения синхронных последовательностных схем* гласят, что схема является синхронной последовательной схемой, если ее элементы удовлетворяют следующим условиям:

- каждый элемент схемы является либо регистром, либо комбинационной схемой;
- как минимум один элемент схемы является регистром;
- все регистры тактируются единственным тактовым сигналом;
- в каждом циклическом пути присутствует как минимум один регистр.

Последовательностные схемы, не являющиеся синхронными, называют асинхронными.

Триггер является самой простой синхронной последовательной схемой с двумя состояниями {0,1}. У него есть один вход данных *D*, один вход тактового сигнала *CLK*, один выход *Q*.

Функциональное описание D-триггера заключается в том, что его следующим состоянием является значение входа *D*, а значение выхода *Q* является текущим состоянием (**рис. 3.20**).

Мы часто будем обозначать текущее состояние переменной *S*, а следующее состояние переменной — *S′*, то есть *S′* обозначает следующее состояние, а не инверсию. Временные диаграммы последовательностных схем будут рассмотрены в **разделе 3.5**.

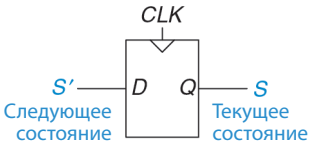


Рис. 3.20 **Текущее и следующее состояния триггера**

Два других вида синхронных последовательностных схем – конечные автоматы и конвейеры. Они будут рассмотрены в этой главе дальше в этой главе.

Пример 3.5 СИНХРОННЫЕ ПОСЛЕДОВАТЕЛЬНОСТНЫЕ СХЕМЫ

Какие из приведенных на **рис. 3.21** схем являются последовательностными синхронными схемами?

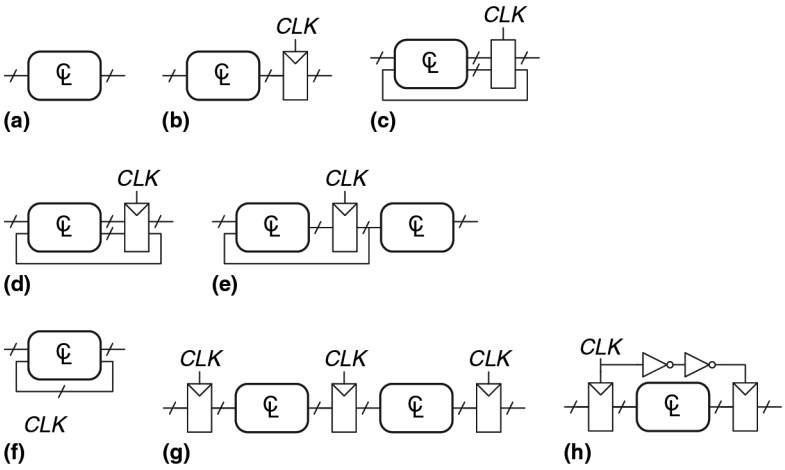


Рис. 3.21 **Примеры схем**

Решение Схема (a) является комбинационной, а не последовательностной, так как в ней отсутствуют регистры. (b) – простая последовательностная схема, так как в ней нет обратной связи. (c) не является ни комбинационной, ни последовательной синхронной схемой, так как она содержит защелку, которая не является ни регистром, ни комбинационной схемой. (d) и (e) – синхронные последовательные логические схемы; они являются двумя классами конечных автоматов, которые будут обсуждаться в **разделе 3.4**. (f) – ни комбинационная, ни синхронная последовательностная, так как у нее есть циклический путь с выхода комбинационной схемы на ее вход, при этом в тракте обратной связи отсутствует регистр. (g) является синхронной последовательной схемой в виде конвейера, который мы изучим в **разделе 3.6**. (h) не является, строго говоря, синхронной последовательностной схемой, так как тактовый сигнал второго регистра отличен от первого из-за задержки, возникающей из-за двух инверторов.

3.3.3. Синхронные и асинхронные схемы

Теоретически из-за отсутствия временных ограничений, накладываемых на систему регистрами, управляемыми тактовыми сигналами, при разработке асинхронных схем разработчик обладает большей свободой, чем при разработке синхронных. Таким же образом, как аналоговые схемы менее формализованы по сравнению с цифровыми, из-за того, что в аналоговых схемах могут использоваться произвольные напряжения, асинхронные схемы менее формализованы, чем синхронные, так как обратная связь в них может быть любой. Но оказывается, что синхронные схемы разрабатывать и использовать проще, чем асинхронные, так же как цифровые схемы проще разрабатывать, чем аналоговые. Несмотря на многолетние научные исследования асинхронных схем, почти все современные цифровые схемы являются синхронными.

Асинхронные схемы иногда используются для связи между собой систем с разными тактовыми сигналами или для считывания значений со входов в произвольное время, так же как аналоговые схемы необходимы для взаимодействия с реальным миром аналоговых (непрерывных) напряжений. Более того, среди разработок в области асинхронных схем есть действительно выдающиеся, некоторые из них могут также улучшить характеристики синхронных схем.

3.4. Конечные автоматы

Последовательностные логические схемы могут быть изображены в форме, представленной на **рис. 3.22**.

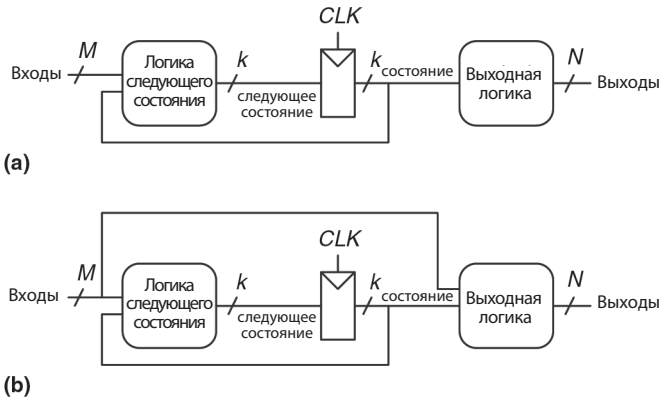


Рис. 3.22 Конечные автоматы: (a) автомат Мура, (b) автомат Мили

Такие формы описания последовательностных схем называются *конечными автоматами* (*КА*). Они получили свое название из-за того, что схема с k-регистрами может находиться в одном из 2^k состояний, то есть в конечном количестве состояний. Любой КА характеризуется M входами,

N выходами и k бит состояний. На вход КА также подается тактовый сигнал и, возможно, сигнал сброса. КА состоит из двух блоков комбинационной логики: логики перехода в *следующее состояние* и *выходной логики*, а также из регистра, в котором хранится текущее состояние. По переднему фронту каждого тактового импульса автомат переходит в следующее состояние, которое определяется текущим состоянием и значениями на входах. Существует два основных класса конечных автоматов, которые отличаются своими функциональными описаниями. В *автомате Мура* выходные значения зависят лишь от текущего состояния, в то время как в *автомате Мили* выход зависит как от текущего состояния, так и от значений на входах. Конечные автоматы представляют собой систематический способ разработки синхронных последовательностных схем по заданному функциональному описанию. Этот метод будет описан ниже, а сейчас мы рассмотрим простой пример.

Автоматы Мура и Мили названы в честь своих изобретателей, ученых, разработавших теорию автоматов и математическую базу для них в фирме Bell Labs.

Эдвард Форест Мур (1925–2003) — не путайте с основателем компании Intel Гордоном Муром — опубликовал свою первую статью «Gedanken-experiments on Sequential Machines» («Мысленные эксперименты с последовательностными автоматами») в 1956 году.

Джорджд Мили (1927–2010) опубликовал «Method of Synthesizing Sequential Circuits» («Метод синтеза последовательностных схем») в 1955 году. Впоследствии он написал первую операционную систему для компьютера IBM 704, работая в Bell Labs. Позже он перешел на работу в Гарвардский университет.

3.4.1. Пример разработки конечного автомата

Для того чтобы проиллюстрировать процесс разработки конечного автомата, рассмотрим проблему создания контроллера светофора для загруженного перекрестка в студенческом городке. Студенты-инженеры гуляют по Академической улице, на которой расположены учебные корпуса и общежитие. У них нет времени читать про конечные автоматы, и они не смотрят под ноги во время ходьбы. Футболисты носятся между спортзалом и столовой по Беговой улице. Они гоняют мяч туда-сюда и тоже не смотрят под ноги. Несколько студентов уже получили серьезные травмы на перекрестке, и декан попросил Бена Битдидла установить светофор, пока не произошел инцидент с летальным исходом.

Бен решил справиться с проблемой с помощью конечного автомата. Он установил два датчика движения, T_A и T_B, на Академической и Беговой улицах соответственно. Каждый датчик выдает единицу, если студенты присутствуют на улице, и ноль, если никого нет. Он также установил два светофора для управления движением, L_A и L_B. Каждый светофор получает входной цифровой сигнал, определяющий, каким светом он должен светить: красным, желтым или зеленым. Следовательно, у КА есть два входа, T_A и T_B, и два выхода, L_A и L_B. Перекресток с двумя светофорами и датчиками показан на **рис. 3.23**. Бен подает на контроллер светофоров тактовые импульсы раз в 5 секунд. По переднему фронту каждого импульса цвет светофора может измениться в зависимости от показаний датчиков движения. Также присутствует кнопка сброса, чтобы техники

могли сбрасывать контроллер после подачи питания в известное исходное состояние. На **рис. 3.24** автомат изображен в виде «черного ящика».

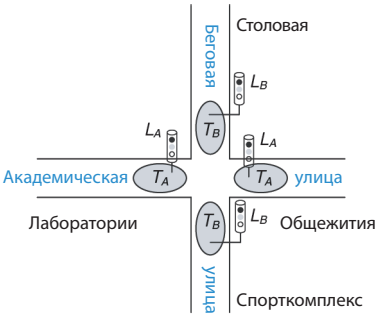


Рис. 3.23 Карта кампуса

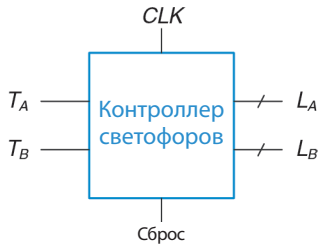


Рис. 3.24 Конечный автомат как «черный ящик»

Следующий шаг Бена Битдидла – сделать эскиз *диаграммы переходов* (или графа), показанный на **рис. 3.25**, на котором приведены все возможные состояния системы и переходы между ними.

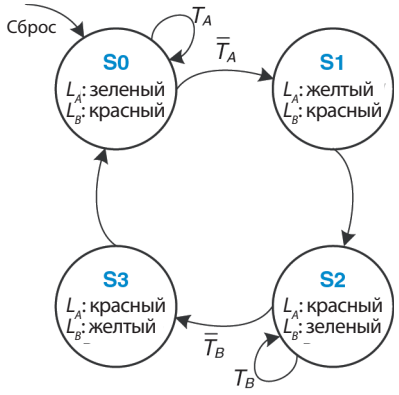


Рис. 3.25 Таблица переходов

После сброса светофор горит зеленым на Академической улице и красным – на Беговой. Каждые 5 секунд контроллер анализирует движение и решает, что делать дальше. Если движение присутствует на Академической улице, то цвет не меняется. Как только Академическая улица освобождается, на ее светофоре 5 секунд горит желтый, затем загорается красный, а на Беговой – зеленый. Аналогично зеленый свет на Беговой улице сохраняется до тех пор, пока улица не станет свободной, затем светофор переключается на желтый, а потом – на красный.

Кружки на диаграмме переходов обозначают состояния, а дуги со стрелками между ними – переходы между этими состояниями. Переходы осуществляются по переднему фронту тактового импульса. Мы не будем изображать тактовый сигнал на диаграмме, так как он всегда

присутствует в синхронных логических схемах. Более того, тактовый сигнал лишь определяет, когда случится переход, тогда как диаграмма определяет, какой именно переход произойдет. Стрелка, обозначенная как сброс, указывает на переход извне в состояние S0, показывая то, что система перейдет в это состояние сразу после сброса, независимо от того, в каком она была состоянии до этого. Если присутствует несколько стрелок, выходящих из некоторого состояния, то эти стрелки подписывают, чтобы показать, какой входной сигнал вызвал этот переход. Например, система находится в состоянии S0. Система останется в состоянии S0, если $T_A = 1$, и перейдет в состояние S1, если $T_A = 0$. Если из этого состояния выходит только одна стрелка, это означает, что такой переход произойдет вне зависимости от состояния входов. Например, из состояния S1 система всегда будет переходить в состояние S2, когда L_A – красный, а L_B – зеленый.

На основе этой диаграммы переходов Бен Битдидл записал таблицу переходов (**табл. 3.1**), которая отражает, каким должно быть следующее состояние S', соответствующее текущему состоянию и входным сигналам. Заметим, что в таблице используются символы X, означающие, что следующее состояние не зависит от текущего входа. Также заметим, что сигнал сброса исключен из этой таблицы. Вместо этого мы использовали сбрасываемые триггеры, которые переходят в состояние S0 сразу после сброса, независимо от данных на входе.

Табл. 3.1. Таблица переходов

Текущее состояние S	Входы T_A	T_B	Следующее состояние S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Заметим, что состояния обозначаются как S0, S1 и т. д. S_0, S_1 – обозначения с индексами – являются битами двоичного числа, соответствующего некоторому состоянию.

Диаграмма переходов абстрактна в том смысле, что она использует состояния, обозначенные как {S0, S1, S2, S3}, и выходы, обозначенные как {красный, желтый, зеленый}.

Для построения реальной схемы состояниям и выходам должны быть поставлены в соответствие двоичные коды.

Бен выбрал простое кодирование (**табл. 3.2** и **3.3**). Каждое состояние и каждое выходное значение закодировано двумя битами: $S_{1:0}$, $L_{A1:0}$ и $L_{B1:0}$.

Таблица 3.2 Кодирование состояний

Состояние	Значение $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Таблица 3.3 Кодирование выходов

Выход	Значение $L_{1:0}$
Зеленый	00
Желтый	01
Красный	10

Бен переписал таблицу переходов, используя двоичное кодирование, как показано в **табл. 3.4**. Эта таблица является таблицей истинности, определяющей логику следующего состояния. Она определяет следующее состояние S' как функцию входов и текущего состояния.

Таблица 3.4 Таблица переходов с двоичным кодированием

Текущее состояние		Входы		Следующее состояние	
S_1	S_0	T_A	T_B	S'	S'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Анализ этой таблицы позволяет легко записать логическую функцию для определения следующего состояния в *совершенной дизъюнктивной нормальной форме* (СДНФ):

$$S'_1 = \overline{S}_1 S_0 + \overline{S}_1 S_0 \overline{T}_B + S_1 \overline{S}_0 T_B;$$
$$S'_0 = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B. \quad (3.1)$$

Уравнения могут быть упрощены при помощи карт Карно, но часто это проще сделать в голове, внимательно изучив уравнения. Например, члены $\overline{T}_B$ и T_B в выражении для S'_1, очевидно, сокращаются. Следовательно, S'_1 сокращается до операции исключающего ИЛИ. Уравнения **(3.2)** являются результатом упрощения уравнений **(3.1)**.

$$S'_1 = S_1 \oplus S_0.$$
$$S'_0 = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B. \quad (3.2)$$

Подобным образом Бен разработал таблицу выходов (**табл. 3.5**), определяя, каким должен быть выход для каждого состояния. Затем он

снова составил и упростил логические выражения для выходов. Например, $L_{A1} = 1$ в строках, где истинно выражение $S_1 = 1$.

$$\begin{aligned} L_{A1} &= S_1. \\ L_{A0} &= \overline{S}_1 S_0. \\ L_{B1} &= \overline{S}_1. \\ L_{B0} &= S_1 S_0. \end{aligned} \tag{3.3}$$

Таблица 3.5 Таблица выходов

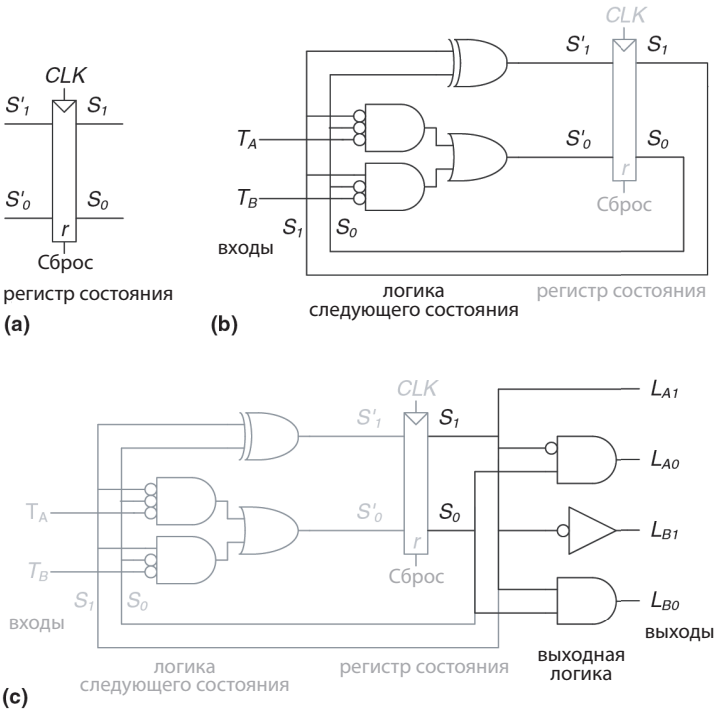
Текущее состояние		Выходы			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Наконец, Бен разработал автомат Мура в форме, приведенной на **рис. 3.22 (a)**. Сначала он изобразил 2-разрядный регистр состояний, как показано на **рис. 3.26 (a)**. По каждому переднему фронту тактового сигнала регистр состояний фиксирует следующее состояние $S'_{1:0}$, и, таким образом, оно становится текущим состоянием $S_{1:0}$. Регистр состояний получает сигнал синхронного или асинхронного сброса для инициализации КА после подачи питания. Затем, основываясь на уравнениях **(3.2)**, Бен нарисовал схему определения следующего состояния, которая вычисляют следующее состояние по значению на входах и по текущему состоянию. Эта схема показана на **рис. 3.26 (b)**. Наконец, он по уравнениям **(3.3)** нарисовал схему (**рис. 3.26 (c)**), которая вычисляет значения на выходах автомата по текущему состоянию.

На **рис. 3.27** показана временная диаграмма, иллюстрирующая переход контроллера светофора из одного состояния в другое. На диаграмме показаны сигнал *CLK*, *сброс* (Reset), входы T_A и T_B, следующее состояние S', текущее состояние S и выходы L_A и L_B. Стрелки показывают причинную связь; например, изменение состояния вызывает изменение выходов, а изменение входов вызывает изменение состояния. Пунктирные линии соответствуют переднему фронту сигнала *CLK*, т. е. времени, когда состояние конечного автомата изменяется.

Период тактового сигнала равен 5 секундам, поэтому сигналы светофора могут переключаться максимум раз в 5 секунд. Когда конечный автомат только включен, его состояние неизвестно, это показывают знаки вопроса. Следовательно, система должна быть сброшена для перевода ее в известное состояние. На этой временной диаграмме S незамедлитель-

но сбрасывается в S0, показывая то, что используются триггеры с асинхронным сбросом. В состоянии S0 свет L_A зеленый, а свет L_B красный.



В этой схеме используются несколько элементов И с кружочками на входах. Их можно сделать из элементов И путем подключения инвертора на вход, или же заменить на элементы ИЛИ-НЕ с обычными входами и инверторами либо на другие комбинации элементов. Выбор логических элементов зависит от особенностей используемой технологии.

Рис. 3.26 Схема конечного автомата контроллера светофора

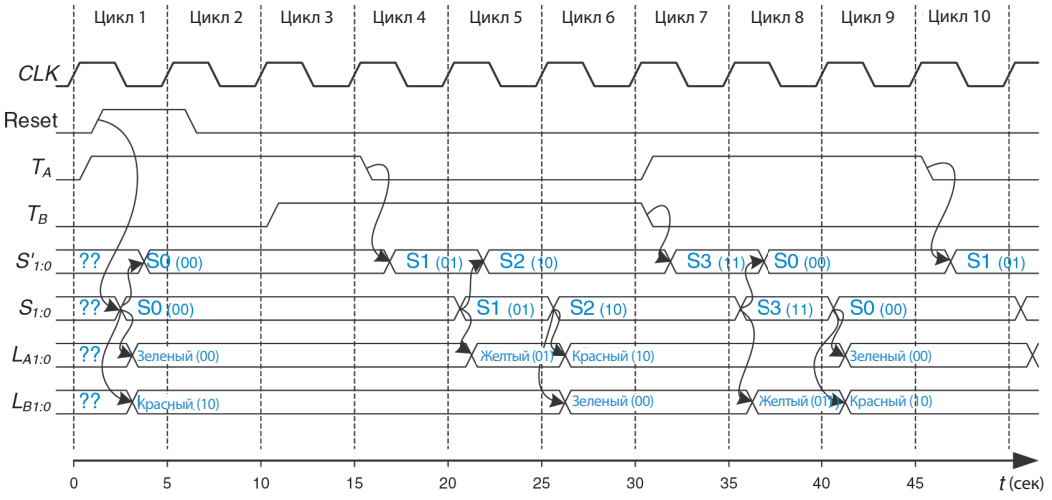


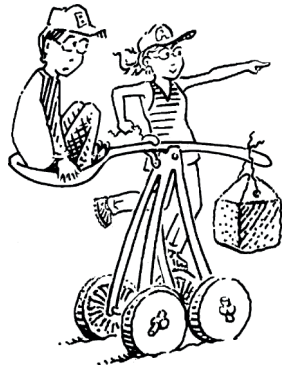
Рис. 3.27 Временная диаграмма работы контроллера светофора

В данном примере движение на Академической улице начинается сразу же. Следовательно, контроллер остается в состоянии S0, оставляя на светофоре L_A зеленый свет, даже если на Беговой улице кто-то появляется. По прошествии 15 секунд поток на Академической улице прекращается, и T_A сбрасывается. Контроллер переходит в состояние S1 по фронту соответствующего тактового импульса и зажигает желтый свет на L_A. Еще через 5 секунд контроллер переходит в состояние S2, в котором на L_A загорается красный, а на L_B – зеленый свет. Контроллер остается в состоянии S2 до тех пор, пока Беговая улица не опустеет. Затем он переходит в состояние S3, зажигая на L_B желтый свет. 5 секунд спустя контроллер переходит в состояние S0, переключая L_B на красный, а L_A – на зеленый свет. Процесс повторяется.

Вопреки ожиданиям, студенты не смотрят на сигналы светофора и продолжают получать травмы. Декан просит Бена Битдидла и Алису Хакер спроектировать катапульту, чтобы бросать студентов через открытые окна лаборатории и общежития, минуя травмоопасное пересечение. Но это тема для другой книги.

3.4.2. Кодирование состояний

В предыдущем примере кодирование состояний и выходов было выбрано произвольно. Выбор другой кодировки привел бы к иной схеме. Основная проблема заключается в том, как определить кодировку, которая потребует наименьшего количества элементов и приведет к наименьшим задержкам в схеме. К сожалению, простого способа найти самую лучшую кодировку не существует, кроме как перепробовать все возможные, что нерационально в случае, если количество состояний велико. Но зачастую возможно найти хорошую кодировку так, чтобы связанные состояния или выходы имели общие биты. При поиске набора возможных кодировок и выбора наиболее рациональной из них часто используются системы автоматизированного проектирования (САПР).



Одно из важных решений в кодировании состояний – выбор между двоичным кодированием (00, 01, 10) и прямым кодированием (001, 010, 100), которое также называется кодированием «1 из N». При *двоичном кодировании*, как в примере с контроллером светофора, каждому состоянию ставится в соответствие двоичное число (номер этого состояния). Так как K двоичных чисел можно записать в $\log_2 K$ разрядах, системе с K состояниями нужно всего $\log_2 K$ бит состояния.

В *прямом кодировании* для каждого состояния используется один бит состояния. По-английски это называется *one-hot-кодированием*, потому что только один разряд будет «горячим», то есть только в одном из разрядов содержится логическая единица в любой момент времени. Например, у КА с прямым кодированием и тремя состояниями коды состояний будут 001, 010 и 100. Каждый бит состояния хранится в триггере; таким образом, прямое кодирование требует

большего количества триггеров, чем двоичное. Но при использовании прямого кодирования схема определения следующего состояния и схема формирования выходных сигналов часто упрощается; таким образом, требуется меньше логических элементов. Наилучший выбор кодирования зависит от особенностей конкретного автомата.

Пример 3.6 КОДИРОВАНИЕ СОСТОЯНИЙ КОНЕЧНОГО АВТОМАТА

У счетчика с делением на N есть один выход, а входов нет. Выход Y находится в состоянии высокого уровня сигнала в течение одного периода каждые N периодов тактового сигнала. Другими словами, выход делит тактовую частоту на N.

На рис. 3.28 приведена временная диаграмма и диаграмма переходов для счетчика-делителя на 3. Нарисуйте схему такого счетчика с использованием двоичного и прямого кодирований.

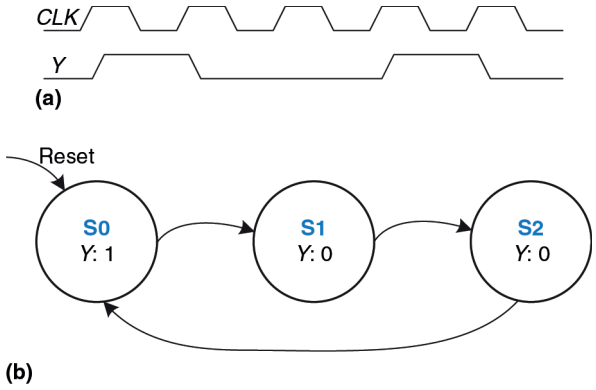


Рис. 3.28 Счетчик-делитель на 3: (a) временная диаграмма, (b) диаграмма переходов

Решение В табл. 3.6 и 3.7 показаны абстрактные таблицы переходов между состояниями и выхода до кодирования.

Таблица 3.6 Кодирование переходов счетчика-делителя на 3

Текущее состояние	Следующее состояние
S0	S1
S1	S2
S2	S0

Таблица 3.7 Кодирование выходов счетчика-делителя на 3

Текущее состояние	Выход
S0	1
S1	0
S2	0

В табл. 3.8 сравниваются двоичное и прямое кодирования для трех состояний.

В двоичном кодировании используются два разряда. Таблица 3.9 является таблицей переходов для этого кодирования. Обратите внимание, что входы отсутствуют; следующее состояние зависит лишь от текущего состояния. Составление таблицы значений на выходе схемы мы оставим читателю в качестве до-

машнего задания. Из этих таблиц легко получить выражения для выхода и для следующего состояния:

$$S'_1 = \overline{S}_1 S_0,$$
$$S'_0 = \overline{S}_1 \overline{S}_0; \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0. \tag{3.5}$$

Таблица 3.8 Двоичное и прямое кодирования счетчика-делителя на 3

Состояние	Кодирование 1-в-1			Двоичное кодирование	
	S_2	S_1	S_0	S'_1	S'_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Таблица 3.9 Таблица переходов с двоичным кодированием

Текущее состояние		Следующее состояние	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

При прямом кодировании используется 3 бита состояния. **Таблица 3.10** – таблица переходов для этого кодирования, а таблицу значений на выходе схемы мы также оставим читателю для самостоятельного выполнения. Выражения для значений на выходе схемы и для следующего состояния будут следующими:

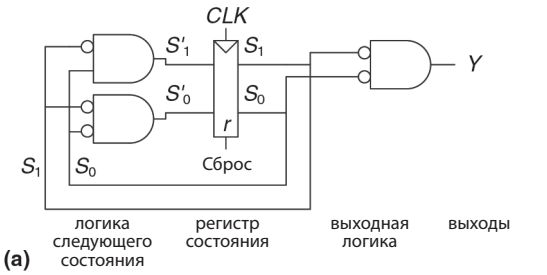
$$S'_2 = S_1,$$
$$S'_1 = S_0, \tag{3.6}$$
$$S'_0 = S_2;$$

$$Y = S_0. \tag{3.7}$$

Таблица 3.10 Таблица переходов с прямым кодированием

Текущее состояние			Следующее состояние		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

На **рис. 3.29** показаны схемы для каждого из двух вариантов. Заметим, что аппаратная реализация схемы при двоичном кодировании может быть оптимизирована путем использования одного элемента для Y и S'_0. Обратите также внимание на то, что при использовании прямого кодирования для инициализации автомата в состояние S0 в момент сброса необходимо использовать триггеры со входами *сброса и установки* (resettable and settable). Выбор наилучшей реализации зависит от относительной стоимости элементов и триггеров, но прямое кодирование обычно предпочтительнее в этом конкретном примере.



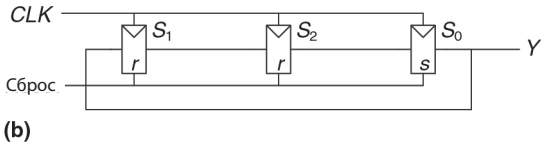


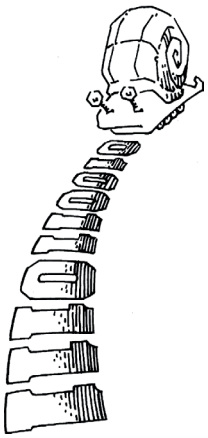
Рис. 3.29 Схемы счетчика-делителя на 3 с двоичным (a) и прямым (b) кодированиями

Еще одной разновидностью прямого кодирования является *one-cold* – кодирование, когда бит, соответствующий состоянию системы в текущий момент, сброшен, в то время как остальные биты установлены: 110, 101, 011.

3.4.3. Автоматы Мура и Мили

До сих пор мы рассматривали примеры автоматов Мура, выходы в которых зависят только от состояния системы. Поэтому на диаграммах переходов для автоматов Мура значения выходов пишутся внутри кружков. Вспомним, что автоматы Мили очень похожи на автоматы Мура, но значения на их выходах могут зависеть от значений на входах таким же образом, как они зависят от текущего состояния автомата. Поэтому на диаграммах переходов для автоматов Мили значения выходов пишутся над стрелками. В блоке комбинационной логики, который вычисляет выходные значения, используются значения текущего состояния и входов, как показано на **рис. 3.22 (b)**.

Простым способом запомнить разницу между двумя типами конечных автоматов состояний является тот факт, что у автомата Мура обычно больше (Moore – more) состояний, чем у автомата Мили, решающего ту же задачу.



Пример 3.7 СРАВНЕНИЕ АВТОМАТОВ МУРА И МИЛИ

У Алисы есть улитка-робот с автоматом с «мозгами» в виде конечного автомата. Улитка ползает слева направо по перфоленте (перфорированные бумажные ленты активно использовались в вычислительной технике в 80-х гг.), содержащей последовательность нулей и единиц. По каждому тактовому импульсу улитка переползает на следующий бит.

Улитка улыбается, если последовательность из двух последних бит, через которые она переползла, равна 01. Разработайте автомат, определяющий, когда улитке нужно улыбнуться. На вход A поступает значение бита под считывающим устройством улитки. На выходе Y устанавливается логическая единица, когда улитка улыбается. Сравните реализации на автоматах Мура и Мили. Нарисуйте временные диаграммы для каждого автомата; изобразите на них вход, состояние и выход; улитка проползает последовательность 0100110111.

Решение Для автомата Мура требуется три состояния, как показано на **рис. 3.30 (a)**. Убедитесь в том, что диаграмма переходов изображена верно, в частности объясните, почему присутствует стрелка из S2 в S1, когда на входе 0.

В отличие от автомата Мура, автомату Мили требуется всего два состояния, как показано на **рис. 3.30 (b)**. Каждая стрелка подписана по принципу A/Y. A – это значение входа, которое вызвало переход, а Y – это соответствующий выходной сигнал.

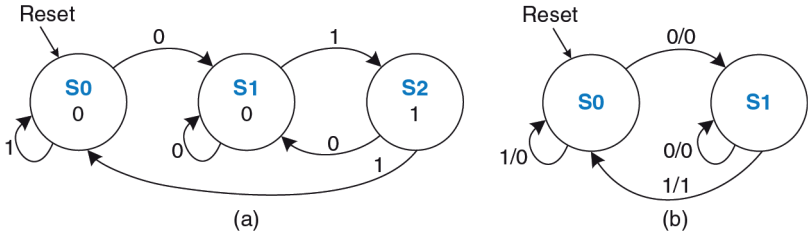


Рис. 3.30 Диаграммы переходов КА: (a) автомат Мура, (b) автомат Мили

В **табл. 3.11** и **3.12** показана диаграмма переходов и таблица состояний выходов для автомата Мура. Автомату Мура потребуется, как минимум, два бита состояния. Мы будем использовать двоичное кодирование: S0 = 00, S1 = 01, S2 = 10. **Таблицы 3.13** и **3.14** являются результатом представления **табл. 3.11** и **3.12** с таким кодированием.

Следовательно, значение следующего состояния и значение выхода для этого состояния ни на что не влияют (X) (не показано в таблицах). Мы пользуемся тем, что это состояние нам безразлично, для упрощения выражений.

Далее составим по этим таблицам выражения для следующего состояния и для выхода. Заметим, что эти выражения упрощены с учетом того, что состояния 11 не существует.

$$S_1' = S_0 A; \qquad S_0' = \overline{A}. \tag{3.8}$$

$$Y = S_1. \tag{3.9}$$

Таблица 3.11 Таблица переходов автомата Мура

Текущее состояние S	Вход A	Следующее состояние S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Таблица 3.12 Таблица выходов автомата Мура

Текущее состояние S	Выход Y
S0	0
S1	0
S2	1

Таблица 3.13 Таблица переходов автомата Мура с кодированием состояний

Текущее состояние S_1	Текущее состояние S_0	Вход A	Следующее состояние S'_1	Следующее состояние S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Таблица 3.14 Таблица выходов автомата Мура с кодированием состояний

Текущее состояние S_1	Текущее состояние S_0	Выход Y
0	0	0
0	1	0
1	0	1

Таблица 3.15 – сводная таблица переходов и выходов для автомата Мили. Автомату Мили необходим только один бит состояния. Будем использовать двоичное кодирование: S0 = 0 и S1 = 1. Преобразуем **табл. 3.15** в **табл. 3.16**, используя такое кодирование.

Таблица 3.15 Таблица переходов и выходов автомата Мили

Текущее состояние S	Вход A	Следующее состояние S'	Выход Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Таблица 3.16 Таблица переходов и выходов автомата Мили с кодированием состояний

Текущее состояние S_0	Вход A	Следующее состояние S_0'	Выход Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

По этим таблицам составим выражения для следующего состояния и для выхода:

$$S_0' = \overline{A}; \tag{3.10}$$

$$Y = S_0 A. \tag{3.11}$$

Схемы автоматов Мили и Мура представлены на **рис. 3.31**. Временные диаграммы для каждого из них изображены на **рис. 3.32**.

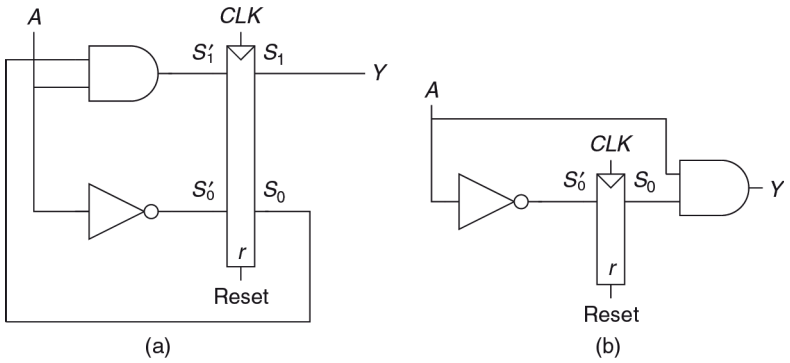


Рис. 3.31 Схемы КА: (a) Мура, (b) Мили

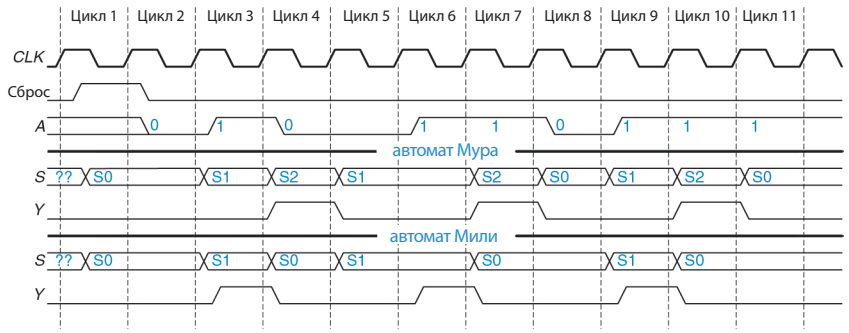


Рис. 3.32 Временные диаграммы автомата Мура и автомата Мили

Каждый из автоматов проходит через разную последовательность состояний. Более того, выход автомата Мили опережает выход автомата Мура на один период, так как он реагирует на вход, а не ждет изменения состояния. Если на выходе автомата Мили поставить триггер, добавив тем самым задержку, то по временным параметрам такая схема станет эквивалентной автомату Мура. Когда будете выбирать тип автомата для вашего проекта, подумайте, в какой момент вы хотите видеть реакцию выходов.

3.4.4. Декомпозиция конечных автоматов

Разработка сложных конечных автоматов часто упрощается, если их можно разделить на несколько более простых автоматов, взаимодействующих друг с другом таким образом, что выход одних автоматов является входом других. Такое применение принципов иерархической организации и модульного проектирования называется *декомпозицией* конечных автоматов.

Пример 3.8 МОДУЛЬНЫЕ И НЕМОДУЛЬНЫЕ КОНЕЧНЫЕ АВТОМАТЫ

Модифицируйте контроллер светофора из **раздела 3.4.1** так, чтобы в нем появился режим «парада». В этом режиме светофор на Беговой улице остается зеленым, когда команда и зрители идут на футбольные игры разрозненными группами. У контроллера появляются еще два входа: P и R. Получая сигнал P, контроллер хотя бы на один цикл входит в режим парада, а получая сигнал R – хотя бы на один цикл выходит из этого режима. Находясь в режиме парада, контроллер проходит свою обычную последовательность переключений до тех пор, пока L_B не станет зеленым, а затем остается в этом состоянии до тех пор, пока режим парада не закончится.

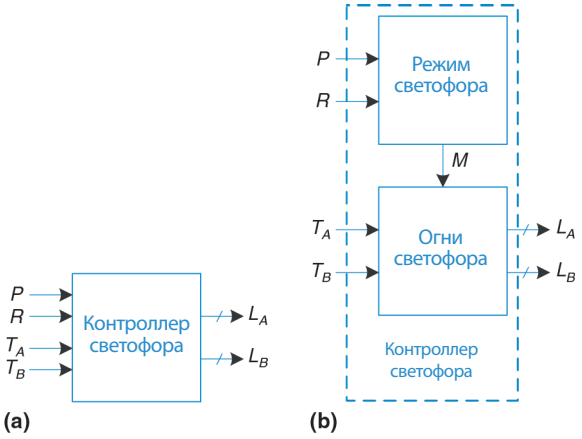


Рис. 3.33 (a) Немодульная и (b) модульная модели КА модифицированного контроллера светофора

Сначала разработаем диаграмму переходов для одного-единственного КА, как показано на **рис. 3.33 (a)**. Затем разработаем диаграмму переходов для двух взаимодействующих КА, как показано на **рис. 3.33 (b)**. Автомат выбора режима устанавливает выход M в единицу, когда он переходит в режим парада. Автомат управления световыми сигналами управляет светофорами в зависимости от M и датчиков движения T_A и T_B.

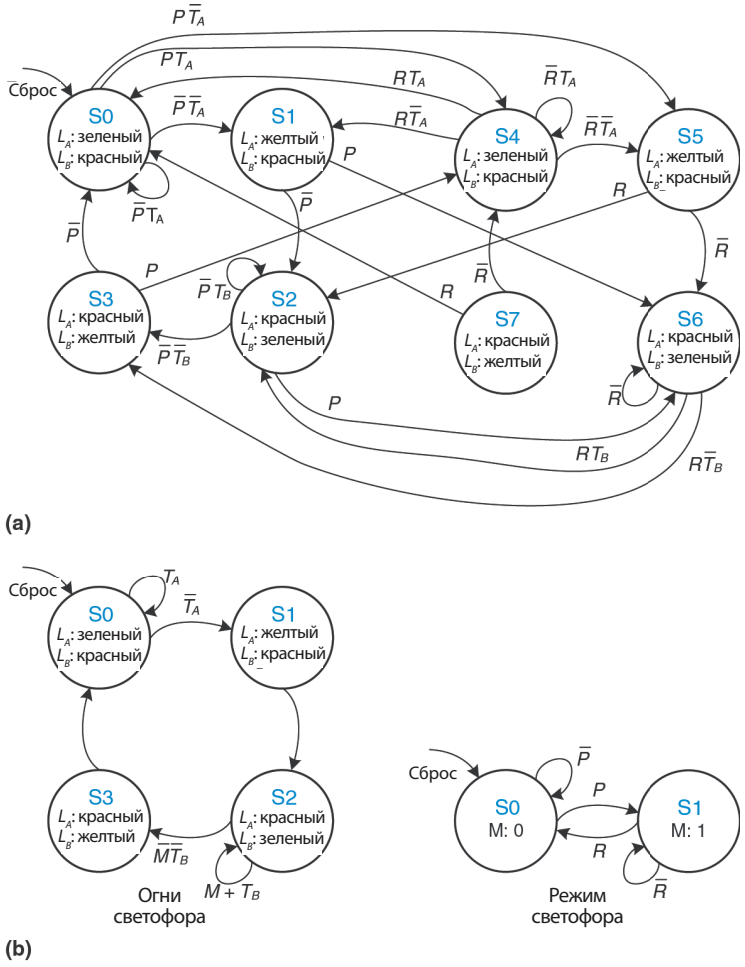


Рис. 3.34 Диаграммы переходов: (a) немодульная, (b) модульная

Решение На **рис. 3.34 (a)** представлена реализация с одним-единственным автоматом. Состояния S0–S3 отвечают за нормальный режим работы, а состояния S4–S7 – за режим парада. Две половины диаграммы практически идентичны, за исключением того, что в режиме парада КА остается в состоянии S6, включая зеленый свет на Беговой улице. Входы P и R управляют переходами между этими двумя половинами. Такой автомат слишком сложный и тяжелый в разработке. На **рис. 3.34 (b)** представлена модульная реализация КА. У КА

выбора режима будет всего два состояния: когда светофор в нормальном режиме и когда – в режиме парада. Автомат управления световыми сигналами модифицирован таким образом, чтобы оставаться в состоянии S2, пока M=1.

3.4.5. Восстановление конечных автоматов по электрической схеме

Восстановление конечных автоматов по электрической схеме является процессом, обратным разработке КА. Этот процесс необходим, например, при рассмотрении проекта с неполной документацией или для реверс-инжиниринга чьей-то системы.

- Проанализируйте схему, возможные состояния входов, выходов и регистра состояний.
- Составьте выражения для определения следующего состояния и для выходов.
- Составьте таблицу выходов и таблицу переходов.
- Вычеркните из таблицы переходов состояния, в которые система никогда не попадает.
- Присвойте имя каждому используемому набору бит-состояний.
- Переработайте таблицы выходов и переходов, используя эти обозначения.
- Разработайте диаграмму переходов.
- Опишите словами то, что делает автомат.

На последнем шаге не бойтесь развернуто описывать цели и функции автомата, чтобы избежать простого переформулирования каждого перехода из диаграммы переходов.

Пример 3.9 ВОССТАНОВЛЕНИЕ КА ПО ЕГО СХЕМЕ

Алиса Хакер приехала домой, но в ее кодовом замке заменили проводку, и ее старый код больше не работает. К замку приколот лист бумаги со схемой, которая приведена на **рис. 3.35**.

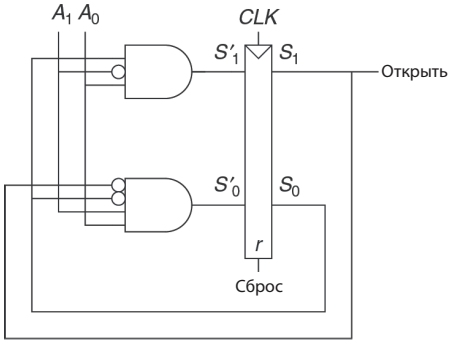


Рис. 3.35 Схема автомата из примера 3.9

Алиса полагает, что схема может быть конечным автоматом, и решает восстановить диаграмму переходов, чтобы узнать, поможет ли ей это попасть внутрь.

Решение Алиса начинает изучать схему. Входом является $A_{1:0}$, а выходом – событие открытия двери. Биты состояний уже обозначены на **рис. 3.35**. Это автомат Мура, так как выходы зависят только от битов состояния. Прямо по схеме она записывает выражения для следующего состояния и для выхода:

$$\begin{aligned} S_1' &= S_0\overline{A}_1A_0; \\ S_0' &= \overline{S}_1\overline{S}_0A_1A_0; \\ Unlock &= S_1. \end{aligned} \tag{3.12}$$

Затем она составляет таблицы переходов и выходов (**табл. 3.17**, **3.18**) по выведенным уравнениям. Сначала Алиса расставляет единицы (последние два столбца таблицы) по выражениям **(3.12)**, а в остальных местах пишет нули.

Таблица 3.17 Таблица следующих состояний, восстановленная по схеме на рис. 3.35

Текущее состояние		Вход		Следующее состояние	
S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Таблица 3.18 Таблица выходов, восстановленная по схеме на рис. 3.35

Текущее состояние		Выход *Unlock*
S_1	S_0	
0	0	0
0	1	0
1	0	1
1	1	1

Алиса сокращает таблицу путем вычеркивания неиспользуемых состояний и путем комбинирования строк, используя при этом безразличные значения. Состоя-

ние $S_{1:0} = 11$ нигде не встречается в **табл. 3.17** как возможное следующее состояние, поэтому строки с этим состоянием можно вычеркнуть. Для текущего состояния $S_{1:0} = 10$ следующее состояние всегда $S_{1:0} = 00$, независимо от входов. **Таблицы 3.19** и **3.20** являются результатом сокращения исходных таблиц.

Таблица 3.19 Сокращенная таблица следующих состояний

Текущее состояние		Вход		Следующее состояние	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Таблица 3.20 Сокращенная таблица выходов

Текущее состояние		Выход *Unlock*
S_1	S_0	
0	0	0
0	1	0
1	0	1

Она присваивает имена для каждой комбинации битов состояний: S0 – это $S_{1:0} = 00$, S1 – это $S_{1:0} = 01$, а S2 – это $S_{1:0} = 10$. Алиса переписывает **табл. 3.19** и **3.20** в **табл. 3.21** и **3.22**, используя эти обозначения.

Таблица 3.21 Символьная таблица следующих состояний

Текущее состояние *S*	Вход *A*	Следующее состояние *S′*
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Таблица 3.22 Символьная таблица выходов

Текущее состояние *S*	Выход *Y*
S0	0
S1	0
S2	1

По **табл. 3.21** и **3.22** она разрабатывает диаграмму переходов, которая представлена на **рис. 3.36**. Изучив ее, она приходит к выводу, что конечный автомат разблокирует дверь после обнаружения поданных на вход $A_{1:0}$ трех единиц. Затем дверь снова блокируется. Алиса пробует ввести этот код, и дверь открывается!

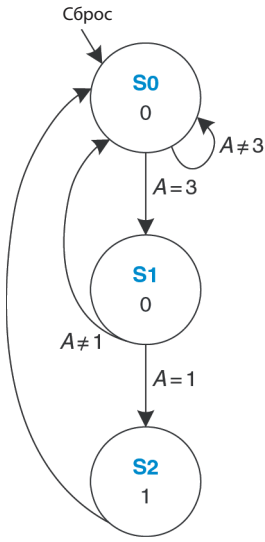


Рис. 3.36. Диаграмма переходов полученного КА

3.4.6. Конечные автоматы: подведение итогов

Конечные автоматы являются мощным инструментом для системного проектирования последовательных схем по техническому заданию. Используйте следующую последовательность действий для создания КА:

- определите входы и выходы;
- разработайте диаграмму переходов;
- для автомата Мура:
  - составьте таблицу переходов;
  - составьте таблицу выходов;
- для автомата Мили:
  - составьте объединенную таблицу выходов и переходов;
- выберите метод кодирования состояний – выбранный метод повлияет на схемотехническую реализацию;
- составьте логические выражения для следующего состояния и для выходной комбинационной схемы;
- разработайте принципиальную схему.

Мы неоднократно будем использовать КА для создания сложных цифровых систем на протяжении всей этой книги.

3.5. Синхронизация последовательностных схем

Вспомните, что триггер копирует сигнал с выхода D на выход Q по переднему фронту тактового сигнала. Этот процесс называется *фиксацией* (sampling) сигнала D по переднему фронту тактового импульса. Поведение триггера корректно, если сигнал на входе D стабилен (равен 0 или 1 и не изменяется) в течение переднего фронта тактового сигнала. Но что произойдет, если сигнал D не будет стабилен во время изменения тактового сигнала?

Эта ситуация аналогична той, которая возникает при спуске затвора фотокамеры. Представьте, что вы пытаетесь

снять прыжок лягушки с плавающего листа кувшинки в озеро. Если вы нажмете на спуск перед прыжком, то на фотографии вы увидите лягушку на листе кувшинки. Если вы нажмете на спуск после прыжка, то на фотографии будет рябь на воде. Но если вы нажмете на спуск во время прыжка, то на фотографии вы увидите смазанное изображение вытянутой вдоль направления прыжка лягушки. Одной из характеристик фотокамеры является *апертурное время*, в течение которого фотографируемый объект должен быть неподвижен, чтобы на фотографии сформировалось его резкое изображение. Аналогично последовательностный элемент имеет апертурное время до и после фронта тактового сигнала, в течение которого его информационные входные сигналы должны быть стабильными, чтобы на выходе триггера сформировался корректный сигнал.

Часть апертурного времени последовательностного элемента до переднего фронта тактового импульса называется *временем предустановки* (setup time), после фронта – *временем удержания* (hold time). Подобно статической дисциплине, которая разрешает использование логических уровней только за пределами запретной зоны, динамическая дисциплина позволяет использовать только те сигналы, которые изменяются вне апертурного времени. При выполнении требований динамической дисциплины мы можем оперировать дискретными единицами времени, которые называются тактовыми циклами, аналогично тому, как мы оперируем с дискретными логическими уровнями 1 и 0. Сигнал может изменяться и колебаться в течение некоторого ограниченного промежутка времени. При выполнении требований динамической дисциплины важно лишь его значение в конце цикла тактового сигнала, когда он уже принял стабильное значение. Следовательно, для описания сигнала A можно использовать его величину $A[n]$ в конце n-го цикла тактового импульса, где n – целое число, вместо его величины $A(t)$ в произвольный момент времени t, где t – действительное число.

Период тактовых импульсов должен быть достаточно большим, чтобы переходные процессы всех сигналов успели завершиться. Это требование ограничивает быстродействие всей системы. В реальных системах тактовые импульсы поступают на входы триггеров неодновременно. Этот разброс по времени, который называется *расфазировкой*, или *разбросом фаз тактового сигнала*, заставляет разработчиков дополнительно увеличивать период тактовых сигналов.

Иногда невозможно удовлетворить требованиям динамической дисциплины, особенно в устройствах сопряжения цифровой системы с реальным миром. Например, рассмотрим схему, ко входу которой подключена кнопка. На кнопку можно нажать как раз во время фронта тактового импульса. Это может привести к возникновению явления, которое называется метастабильностью, при этом триггер оказывается в промежуточном состоянии между 0 и 1, причем переход в корректное логическое состояние (0 или 1) может происходить бесконечно долго.

Решением проблемы асинхронных входов является использование синхронизатора, на выходе которого некорректный логический уровень может появиться с очень малой (но не нулевой) вероятностью.

Эти идеи будут детально рассмотрены в оставшейся части раздела.

3.5.1. Динамическая дисциплина

До сих пор мы рассматривали функциональные спецификации последовательных схем. Вспомните, что синхронные последовательные схемы, такие так триггеры или конечные автоматы, имеют также и временную спецификацию, пример которой показан на **рис. 3.37**.

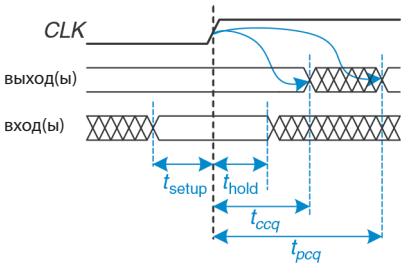


Рис. 3.37 Временная спецификация синхронной последовательной схемы

После перехода 0 → 1 тактового сигнала (переднего фронта тактового импульса) выход (или выходы) схемы может начать изменяться не ранее чем через время t_{ccq} – задержка реакции (Clock-to-Q contamination delay[1]), и должен принять стационарное значение не позднее чем через время t_{pcq} – задержка распространения (Clock-to-Q propagation delay). Эти величины представляют собой наименьшую и наибольшую задержки схемы соответственно. Для того чтобы схема корректно среагировала на сигнал, информационный вход (или входы) схемы должен быть стабильным в течение некоторого *времени предустановки* (setup time) t_{setup} до прихода переднего фронта тактового сигнала и не должен изменяться в течение *времени удержания* (hold time) t_{hold} после прихода переднего фронта тактового сигнала. Сумма времен *предустановки* и *удержания* называется *апертурным временем* схемы. Это общее время, в течение которого информационный входной сигнал должен быть стабилен для его фиксации на выходе.

Динамическая дисциплина требует, чтобы входы синхронной последовательной схемы были стабильны в течение времени предустановки до и времени удержания после переднего фронта тактового импульса. Выполнение этих требований гарантирует, что в процессе фиксации значения информационного входа триггером он не будет изменяться. Поскольку мы будем рассматривать только установившиеся значения входных

[1] В российской, да и зарубежной нормативно-технической документации чаще всего используется только задержка распространения (propagation delay), но указываются ее минимальное и максимальное значения.

сигналов в моменты времени, когда они фиксируются, мы можем считать сигналы дискретными как по уровню, так и по времени.

3.5.2. Временные характеристики системы

За тридцать лет, прошедших со времени, когда семья одного из авторов купила компьютер Apple II+, до момента подготовки этой книги, тактовая частота микропроцессора увеличилась с 1 МГц до нескольких ГГц, более чем в тысячу раз. Это увеличение быстродействия компьютеров частично объясняет революционные изменения, которые благодаря им произошли в обществе.

Периодом тактового сигнала, или *длительностью цикла синхронизации*, T_c, называется промежуток времени между передними фронтами последовательных тактовых импульсов. Обратная величина, $f_c = 1/T_c$, называется *тактовой частотой*. Увеличение тактовой частоты без изменения остальных параметров схемы приводит к увеличению ее производительности. Частота измеряется в герцах (Гц), или в циклах за одну секунду: 1 мегагерц (МГц) = 10^6 Гц и 1 гигагерц (ГГц) = 10^9 Гц.

На **рис. 3.38 (a)** показана характерная структура тракта обработки данных синхронной последовательной схемы, для которой мы рассчитаем период тактового сигнала. По переднему фронту тактового импульса на выходе регистра R1 формируется выходной сигнал (или сигналы) Q1. Эти сигналы поступают на вход блока комбинационной логики, выходные сигналы этого блока поступают на вход (или входы) D2 регистра R2. Как показано на **рис. 3.38 (b)**, выходной сигнал блока может начать изменяться не ранее окончания времени реакции после завершения изменения его входного сигнала и принимает окончательное значение спустя максимальное время задержки распространения от момента установки входного сигнала. Серые стрелки показывают минимальную задержку с учетом регистра R1 и комбинационной логики, а синие – максимальную задержку распространения в тракте регистр R1 – комбинационная логика. Мы анализируем временные ограничения с учетом времен предустановки и удержания второго регистра, R2.

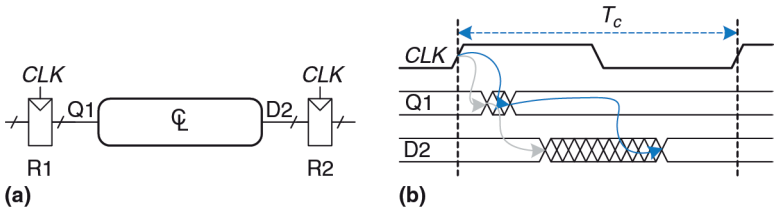


Рис. 3.38 Тракт между регистрами и временная диаграмма

Ограничение времени предустановки

На **рис. 3.39** на временной диаграмме приведена только максимальная задержка в тракте обработки информации; эта задержка обозначена синими стрелками.

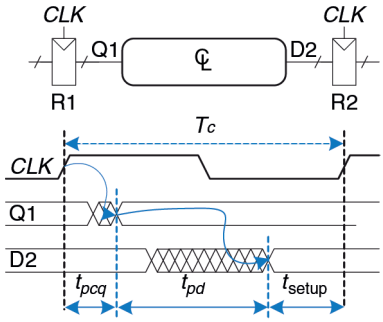


Рис. 3.39 Максимальная задержка для ограничения времени предустановки

Для выполнения ограничения по времени предустановки регистра R2 сигнал D2 должен установиться не позднее, чем за время предустановки фронта следующего тактового импульса. Таким образом, мы можем получить выражение для минимальной длительности периода синхросигнала:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}. \tag{3.13}$$

При разработке коммерческих продуктов период тактового сигнала будущего изделия часто задается из соображений конкурентоспособности руководителем отдела разработок или отделом маркетинга. Более того, задержка распространения сигнала триггером от фронта тактового сигнала до выхода (Clock-to-Q) и время предустановки t_{pcq} и t_{setup} обычно определены производителем. Следовательно, неравенство **(3.13)** следует преобразовать для определения максимальной задержки распространения комбинационной схемы, поскольку обычно именно это – единственный параметр, который может изменять разработчик:

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}). \tag{3.14}$$

Слагаемое в скобках, $t_{pcq} + t_{\text{setup}}$, называется *потерями на упорядочение* (sequencing overhead). В идеальном случае весь период тактового сигнала может быть затрачен на вычисления в комбинационной логике (время t_{pd}). Но потери на упорядочение в триггерах уменьшают это время. Неравенство **(3.14)** называется *ограничением времени предустановки*, или *ограничением максимальной задержки*, поскольку оно зависит от времени предустановки и ограничивает максимальную задержку распространения в комбинационной схеме.

Если задержка распространения в комбинационной схеме слишком велика, то вход D2 может не успеть принять свое установившееся состояние к моменту времени, когда регистр R2 ожидает стабильный сигнал и фиксирует его. Таким образом, R2 может зафиксировать некорректный результат или даже логический уровень сигнала в запретной зоне. В та-

ком случае схема будет работать некорректно. Проблему можно решить увеличением периода тактового сигнала или пересмотром комбинационной схемы с целью добиться меньшей задержки распространения.

Ограничение времени удержания

Регистр R2 на **рис. 3.38 (а)** имеет также *ограничение времени удержания*. Его вход D2 не должен изменяться в течение некоторого времени t_{hold} после переднего фронта тактового импульса.

В соответствии с **рис. 3.40** D2 может измениться через $t_{ccq} + t_{cd}$ после переднего фронта тактового импульса. Следовательно, можно записать:

$$t_{ccq} + t_{cd} \geq t_{hold}. \tag{3.15}$$

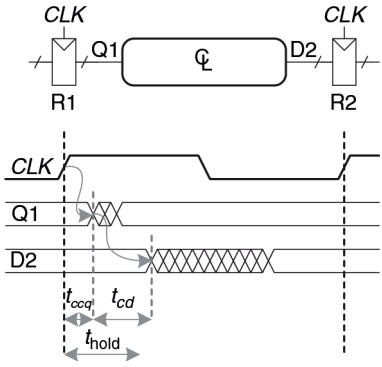


Рис. 3.40 Минимальная задержка для ограничения времени удержания

Как и ранее, характеристики используемого в схеме триггера t_{ccq} и t_{hold} обычно находятся вне влияния разработчика схемы. После простых преобразований мы можем записать неравенство для минимальной задержки комбинационной логической схемы:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq}. \tag{3.16}$$

Неравенство **(3.16)** также называется *ограничением времени удержания*, или *ограничением минимальной задержки*, потому что оно ограничивает минимальную задержку комбинационной схемы.

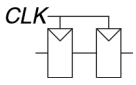


Рис. 3.41 Непосредственное последовательное соединение триггеров

Мы предполагаем, что при соединении логических элементов между собой временные проблемы синхронизации не возникают. В частности, мы считаем, что при непосредственном последовательном соединении двух триггеров, как показано на **рис. 3.41**, проблемы, обусловленные временем удержания, не возникают.

В этом случае вследствие отсутствия комбинационной логики между триггерами $t_{cd} = 0$. При такой подстановке неравенство **(3.16)** сводится к требованию:

$$t_{\text{hold}} \leq t_{ccq}. \tag{3.17}$$

Иными словами, время удержания надежного триггера должно быть меньше, чем его задержка реакции. Часто триггеры разрабатывают так, что $t_{\text{hold}} = 0$, следовательно, неравенство **(3.17)** всегда выполняется. В этой книге, если не указано обратное, мы будем считать такое предположение истинным и игнорировать ограничение времени удержания.

Тем не менее ограничения времени удержания критически важны. Если они нарушаются, то единственным решением является увеличение задержки реакции комбинационной схемы, что требует ее повторной разработки. Такие нарушения, в отличие от нарушений ограничений времени предустановки, не могут быть исправлены изменением периода тактового сигнала. Повторная разработка интегральной микросхемы и производство ее исправленного варианта занимают несколько месяцев и требуют затрат в несколько миллионов долларов при современных технологиях, поэтому к *нарушениям ограничения времени удержания* нужно относиться крайне серьезно.

Краткие выводы к подразделу

Последовательностные схемы имеют ограничения времен предустановки и удержания, которые устанавливают максимальную и минимальную задержки в комбинационной схеме между триггерами. Современные триггеры обычно разработаны так, что минимальная задержка в комбинационной логике равна нулю, то есть триггеры могут быть размещены непосредственно друг за другом. Максимальная задержка ограничивает количество логических элементов, включенных один за другим в критическом пути быстродействующей схемы.

Пример 3.10 ВРЕМЕННОЙ АНАЛИЗ

Бен Битдидл разработал схему, которая показана на **рис. 3.42**. В соответствии со спецификацией компонентов, которые он использует, задержка реакции на тактовый вход-выход триггеров равна 30 пс, а задержка распространения – 80 пс. Они имеют время предустановки 50 пс и время удержания 60 пс. У логических элементов задержка распространения равна 40 пс, задержка реакции – 25 пс. Помогите Бену определить максимальную тактовую частоту его схемы и выяснить, могут ли происходить нарушения ограничения времени удержания в ней. Этот процесс называется временным анализом.

Решение На **рис. 3.43 (a)** приведены временные диаграммы сигналов, которые показывают, когда они могут изменяться. На входы A–D сигнал поступает с регистров, поэтому они могут измениться через короткое время после переднего фронта сигнала CLK.

Критический путь возникает, когда $B = 1$, $C = 0$, $D = 0$ и A изменяется из 0 в 1, что приводит к переключению n1 в 1, X' в 1, Y' в 0, как показано на **рис. 3.43 (b)**. В этот путь входят задержки трех логических элементов. Для оценки задерж-

ки в критическом пути будем считать, что задержка каждого элемента равна задержке распространения. Сигнал Y' должен установиться ранее следующего переднего фронта CLK. Следовательно, минимальная длительность цикла равна

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250 \text{ пс}. \quad (3.18)$$

Максимальная тактовая частота равна $f_c = 1/T_c = 4$ ГГц.

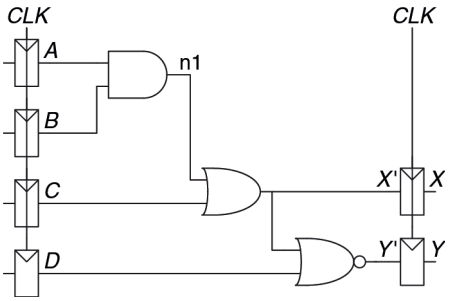


Рис. 3.42 Пример схемы для временного анализа

Короткий (по времени прохождения сигналом) путь возникает, когда $A = 0$ и C переключается в 1, как показано на **рис. 3.43 (c)**.

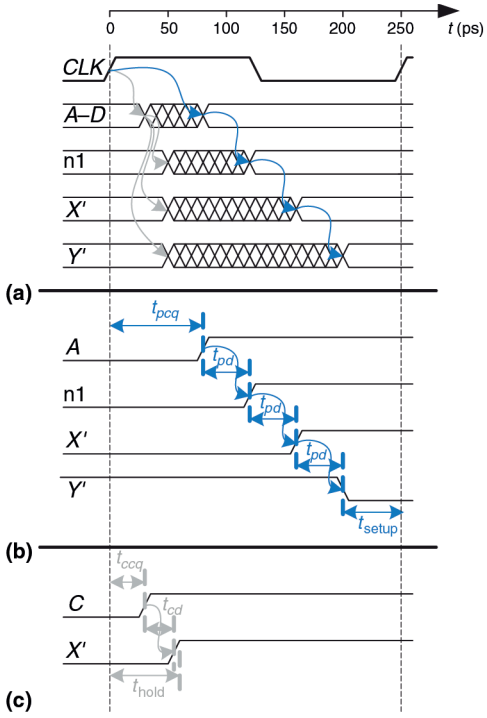


Рис. 3.43 Временная диаграмма: (a) общий случай, (b) критический путь, (c) короткий путь

Для короткого пути будем считать, что каждый логический элемент переключается сразу после завершения задержки реакции. Этот путь включает в себя только один элемент, поэтому переключение может наступить через $t_{ccq} + t_{cd} = 30 + 25 = 55$ пс. Но следует помнить, что время удержания триггера равно 60 пс. Это означает, что сигнал X' обязательно должен быть стабильным в течение 60 пс после переднего фронта тактового сигнала CLK, чтобы триггер смог надежно зафиксировать значение сигнала X'. В этом случае если в течение первого переднего фронта CLK вход $X' = 0$, то триггер должен зафиксировать 0. Но, поскольку X' не поддерживается стабильным в течение времени удержания, действительное значение X будет непредсказуемым. В этой схеме происходит нарушение ограничений времени удержания, и ее поведение непредсказуемо при любой тактовой частоте.

Пример 3.11 ИСПРАВЛЕНИЕ НАРУШЕНИЙ ВРЕМЕНИ УДЕРЖАНИЯ

Алиса Хакер предлагает исправить схему Бена путем добавления буферных элементов, которые будут замедлять прохождение сигнала через короткий путь, как показано на **рис. 3.44**. Буферы имеют такую же задержку, как и остальные логические элементы. Определите максимальную тактовую частоту и проверьте, будут ли возникать проблемы, связанные со временем удержания.

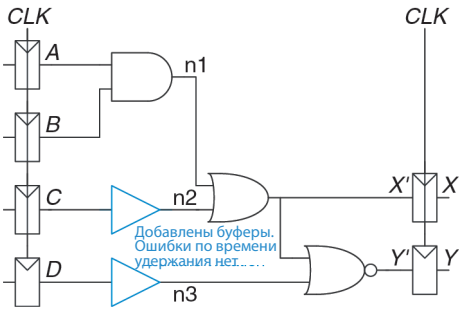


Рис. 3.44 Исправленная схема, в которой отсутствуют нарушения ограничения времени удержания

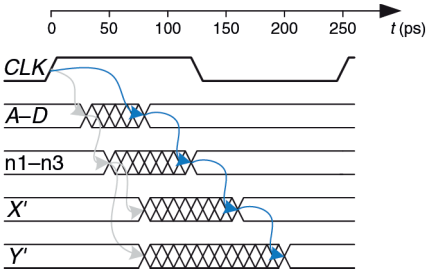


Рис. 3.45 Временная диаграмма схемы с буферами, в которой отсутствуют нарушения ограничения времени удержания

Решение На **рис. 3.45** приведены временные диаграммы, которые показывают, когда сигналы могут изменяться. Критический путь от A до Y не изменился, потому что он не проходит через буферы. Следовательно, максимальная тактовая частота равна, как и ранее, 4 ГГц. При этом время прохождения сигнала через короткий путь будет увеличено на величину минимальной задержки бу-

феров. Теперь X' не изменится в течение $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ пс после фронта тактового сигнала. Таким образом, X' будет стабилен в течение времени удержания 60 пс, то есть схема будет работать правильно.

В этом примере аномально большое время удержания было использовано только для демонстрации сути проблем, связанных со временем удержания. Большинство триггеров разработаны так, что $t_{hold} < t_{ccq}$, это позволяет избежать таких проблем. Но в некоторых высокопроизводительных микропроцессорах, включая Pentium IV, вместо триггеров используется элемент, который называется *импульсная защелка* (pulsed latch). Импульсная защелка ведет себя подобно обычному триггеру, но имеет небольшую задержку распространения тактового сигнала от входа к информационному выходу и большое время удержания. Добавление буферов позволяет часто, но не всегда, устранить проблемы, связанные с ограничением времени удержания, без увеличения времени прохождения сигнала по критическому пути.

3.5.3. Расфазировка тактовых сигналов

В предыдущих разделах предполагалось, что тактовые импульсы поступают на все регистры в одно и то же время. В действительности существует некоторый разброс этого времени. Эта неодновременность прихода передних фронтов тактовых импульсов называется *расфазировкой*. Например, длина проводников, по которым тактовые сигналы поступают на разные регистры, может быть разной, что приводит к разным временам задержки, как показано на **рис. 3.46**.

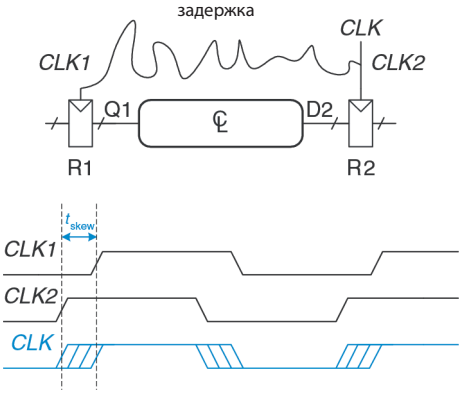


Рис. 3.46 Расфазировка тактовых сигналов, обусловленная задержками в межсоединениях

Шум также приводит к различным задержкам. Использование схем разрешения тактовых сигналов, которое было описано в **разделе 3.2.5**, приводит к их дополнительной задержке. Если в схеме для одних тактовых сигналов используются схемы разрешения, а для других -- нет, то между ними будет существенное рассогласование. На **рис. 3.46** сигнал *CLK*2 будет опережать по времени сигнал *CLK*1 из-за сложного пути тактового сигнала между регистрами. Если трассировка цепи тактового

сигнала будет выполнена по-другому, ситуация может быть противоположной, *CLK*2 будет отставать от сигнала *CLK*1. При выполнении временного анализа мы рассматриваем наихудший случай, что позволяет гарантировать, что схема будет работать при всех условиях.

Учет расфазировки изменяет временную диаграмму, которая была показана на **рис. 3.38**. Модифицированная диаграмма приведена на **рис. 3.47**.

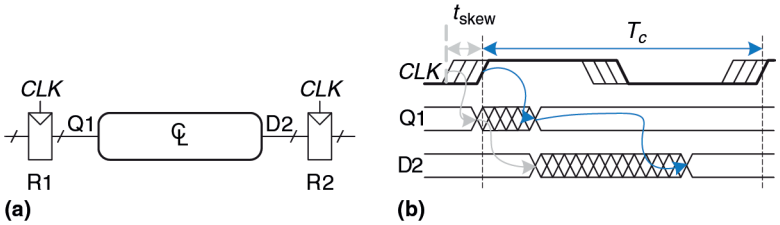


Рис. 3.47 Временная диаграмма с учетом расфазировки тактовых импульсов

Жирной линией показана максимальная задержка тактового сигнала, а тонкие линии показывают, что синхросигнал может появиться на t_{skew} раньше.

Вначале рассмотрим ограничение времени предустановки. Соответствующие диаграммы приведены на **рис. 3.48**.

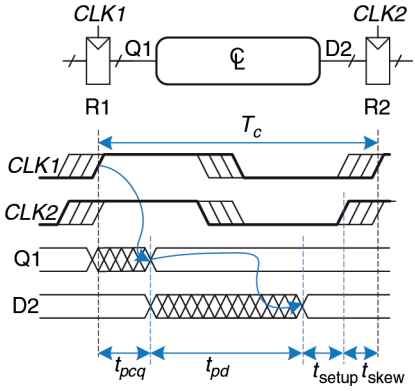


Рис. 3.48 Ограничение времени предустановки с учетом расфазировки тактовых импульсов

В худшем случае на регистр R1 поступает тактовый сигнал с наибольшей задержкой, а на R2 – с наименьшей, что оставляет минимальное время для прохождения данных через комбинационную схему между регистрами.

На вход регистра R2 данные поступают через регистр R1 и комбинационную логику, они должны прийти к стационарному состоянию перед началом их фиксации регистром R2. Следовательно, можно сделать вывод, что

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}; \qquad (3.19)$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq}. \tag{3.20}$$

Далее мы рассмотрим ограничение времени удержания (**рис. 3.49**). В худшем случае на регистр R1 поступает тактовый сигнал с наименьшей задержкой, а на R2 – с наибольшей. Данные могут быстро пройти через регистр R1 и комбинационную логику, но должны поступить на вход регистра R2 не ранее окончания времени удержания после переднего фронта тактового импульса.

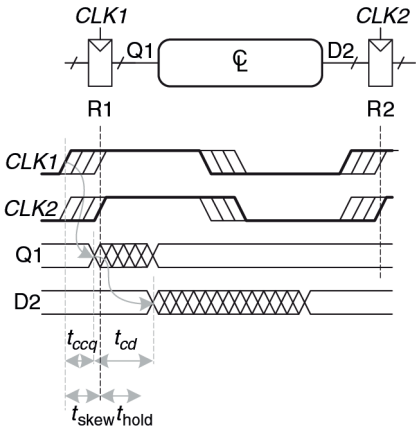


Рис. 3.49 Ограничение времени удержания с учетом расфазировки тактовых импульсов

Таким образом, можно записать:

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}}; \tag{3.21}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq}. \tag{3.22}$$

В итоге расфазировка тактовых импульсов приводит к увеличению как времени предустановки, так и времени удержания. Это, в свою очередь, приводит к росту потерь на упорядочение и уменьшает время, доступное для обработки данных комбинационной схемой. Даже если $t_{\text{hold}} = 0$, пара последовательно соединенных триггеров будет нарушать неравенство **(3.22)**, если $t_{\text{skew}} > t_{ccq}$. Чтобы предотвратить такие серьезные нарушения ограничений времени удержания, разработчик должен ограничивать расфазировку тактовых сигналов. Иногда триггеры специально разрабатывают медленными (время которых t_{ccq} велико), чтобы избежать проблем времени удержания, даже если расфазировка тактовых сигналов существенна.

Пример 3.12 ВРЕМЕННОЙ АНАЛИЗ РАСФАЗИРОВКИ ТАКТОВЫХ СИГНАЛОВ

Выполните задание из **примера 3.10**. Будем считать, что расфазировка тактовых импульсов в системе составляет 50 пс.

Решение Критический путь остается без изменений, но эффективное время предустановки увеличивается из-за расфазировки. Следовательно, минимальный период тактового сигнала равен

$$T_c \geq t_{pcq} + 3_{tpd} + t_{\text{setup}} + t_{\text{skew}} = 80 + 3{,}40 + 50 + 50 = 300 \text{ пс}. \quad (3.23)$$

Максимальная частота тактового сигнала будет $f_c = 1 / T_c = 3{,}33$ ГГц.

Короткий путь также остается без изменений, а время прохождения сигнала по нему равно 55 пс. Эффективное время удержания увеличивается на величину расфазировки до 60 + 50 = 110 пс, что существенно больше 55 пс. Следовательно, в схеме будет нарушено ограничение времени удержания, и она будет некорректно работать при любой частоте тактового сигнала. Напомним, что в этой схеме ограничение времени удержания было нарушено и без расфазировки. Расфазировка тактовых сигналов только ухудшила ситуацию.

Пример 3.13 ИСПРАВЛЕНИЕ НАРУШЕНИЯ ОГРАНИЧЕНИЯ ВРЕМЕНИ

Повторите упражнение из **примера 3.11** при условии, что в системе есть расфазировка тактовых импульсов величиной 50 пс.

Решение Критический путь не изменяется, поэтому максимальная тактовая частота остается равной 3,33 ГГц. Время прохождения сигнала по короткому пути увеличивается до 80 пс. Это все еще меньше, чем $t_{\text{hold}} + t_{\text{skew}} =$ 110 пс, следовательно, в схеме нарушаются ограничения времени удержания. Чтобы решить проблему, в схему следует добавить еще несколько буферов. Поскольку они входят в критический путь, то максимальная тактовая частота уменьшится. В качестве альтернативы можно рассмотреть использование других триггеров с меньшим временем удержания.

3.5.4. Метастабильность

Как было показано ранее, не всегда можно гарантировать, что вход последовательностной схемы будет стабилен в течение апертурного времени, особенно если входной сигнал поступает от внешнего асинхронного источника. Рассмотрим кнопку, подсоединенную ко входу триггера, как показано на **рис. 3.50**. Когда кнопка не нажата, $D = 0$. Когда кнопка нажата, $D = 1$. Можно нажимать кнопку в любой произвольный момент времени по отношению к переднему фронту тактового сигнала. Мы хотим знать сигнал на выходе Q после переднего фронта сигнала CLK. В случае I, когда кнопка нажимается задолго до фронта CLK, $Q = 1$. В случае II кнопка нажимается намного позже фронта CLK, $Q = 0$. Но в случае III, когда кнопка нажимается в промежутке, который охватывает время предустановки перед фрон-

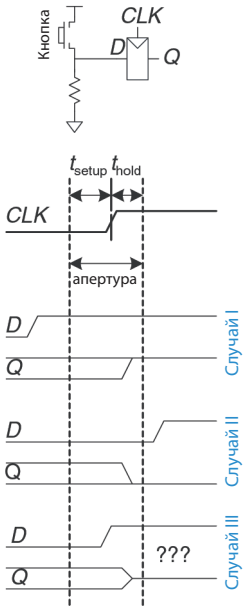


Рис. 3.50 Входной сигнал, который изменяется до, после или в течение апертурного времени

том тактового импульса и время удержания после него, входной сигнал нарушает динамическую дисциплину и выход будет неопределенным.

Метастабильное состояние

Когда состояние информационного входа триггера изменяется в течение апертурного времени, на его выходе Q может на некоторое время появиться напряжение в диапазоне от 0 до V_{DD}, то есть в запретной зоне. Такое состояние называется *метастабильным*. Со временем выход триггера перейдет в *стабильное состояние* 0 или 1. Но *время разрешения*, необходимое для достижения стабильного состояния, не определено.

Рис. 3.51 Стабильное и метастабильное состояния

Метастабильное состояние триггера подобно состоянию шарика на вершине между двумя впадинами, как показано на **рис. 3.51**. Положения во впадинах являются стабильными, поскольку шарик будет находиться в них неограниченно долго при отсутствии внешнего возмущения. Положение на вершине возвышенности называется *метастабильным*, потому что шарик будет находиться в нем только при условии идеальной балансировки. Но поскольку в мире нет ничего совершенного, со временем шарик скатится в одну из впадин. Необходимое для этого время зависит от степени первоначальной балансировки шарика. Каждое бистабильное устройство имеет метастабильное состояние между двумя стабильными.

Время разрешения

Если вход триггера изменяется в произвольный момент цикла тактового сигнала, то время разрешения, t_{res}, необходимое для перехода в стабильное состояние, также является случайной величиной. Если вход изменяется вне апертурного времени, то $t_{res} = t_{pcq}$. Но если произойдет изменение входа в апертурное время, t_{res} может быть существенно больше.

Теория и практика (**раздел 3.5.6**) показывают, что вероятность того, что время разрешения превышает некоторое время t, экспоненциально падает с ростом t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}}, \tag{3.24}$$

где T_c – период тактового сигнала, T_0 и τ – характеристики триггера. Выражение справедливо, только если t намного больше, чем t_{pcq}.

Интуитивно понятно, что отношение T_0 / T_c описывает вероятность того, что вход изменится в неудачное время (то есть в апертурное время); эта вероятность уменьшается с ростом периода тактового сигнала T_c. τ – временная константа, которая показывает, насколько быстро триггер

выходит из метастабильного состояния; она связана с задержкой в перекрестно соединенных логических элементах триггера.

Таким образом, если вход бистабильного устройства, такого как триггер, изменяется в течение апертурного времени, его выход может некоторое время находиться в метастабильном состоянии, прежде чем перейти в стабильное состояние 0 или 1. Время перехода в стабильное состояние не ограничено, потому что для любого конечного времени t вероятность того, что триггер все еще находится в метастабильном состоянии, не равна нулю. Но эта вероятность экспоненциально падает с ростом t. Следовательно, если подождать достаточно долго, намного больше, чем t_{pcq}, то с весьма высокой вероятностью можно ожидать того, что триггер достигнет корректного логического состояния.

3.5.5. Синхронизаторы

Наличие асинхронных входов цифровой системы, которые принимают информацию из внешнего мира, неизбежно. Например, сигналы, которые формирует человек, асинхронны. Такие асинхронные входы, если к ним относиться небрежно, могут привести к появлению метастабильных состояний в системе, что приведет к ее непредсказуемым отказам, которые крайне сложно отследить и исправить. При наличии асинхронных входов разработчик системы должен обеспечить достаточно малую вероятность появления метастабильных напряжений. Смысл слова «достаточно» зависит от контекста. Для сотового телефона один отказ за 10 лет допустим, потому что пользователь может всегда выключить и включить телефон, если он «зависнет». Для медицинского прибора более предпочтительным является один отказ за предполагаемое время существования Вселенной (10^{10} лет). Чтобы гарантировать корректность логических уровней, все асинхронные входы должны пройти через *синхронизаторы*.

Синхронизатор, как показано на **рис. 3.52**, является устройством, на вход которого поступают асинхронный сигнал D и тактовый сигнал CLK. За ограниченное время он формирует выходной сигнал Q, который с очень высокой вероятностью имеет корректный логический уровень. Если вход D стабилен в течение апертурного времени, то выход Q должен принять значение входа. Если D изменяется в течение апертурного времени, то Q может принять значение 0 или 1, но не должен быть метастабильным.

На **рис. 3.53** показано, как из двух триггеров можно построить простой синхронизатор. Триггер F1 фиксирует значение входного сигнала D по переднему фронту тактового сигнала CLK. Если D изменяется в апертурное время, его выход D2 на некоторое время может стать метастабильным. Если период тактового сигнала достаточно велик, то с высокой вероятностью до конца периода D2 придет к корректному логическо-

му уровню. Триггер F2 затем фиксирует D2, который теперь стабилен, и формирует корректный выходной сигнал.

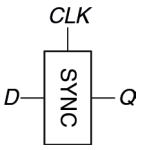


Рис. 3.52 Символ синхронизатора

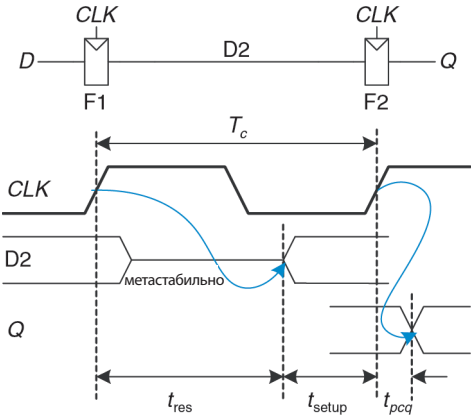


Рис. 3.53 Простой синхронизатор

Мы говорим о сбое синхронизатора, если его выход Q станет метастабильным. Это может произойти, если D2 не успеет прийти в корректное состояние до начала времени предустановки триггера F2, то есть когда $t_{res} > T_c - t_{\text{setup}}$. В соответствии с выражением **(3.24)** вероятность сбоя для одиночного изменения входа в произвольное время равна

$$P(\text{failture}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}}. \tag{3.25}$$

Вероятность сбоя, $P(\text{failure})$, есть вероятность того, что выход Q будет метастабильным после однократного изменения входа D. Если D изменяется один раз за секунду, то вероятность сбоя за одну секунду будет просто $P(\text{failure})$. Но если D изменяется N раз за секунду, то вероятность ошибки за секунду будет в N раз большей:

$$P(\text{failture})/\text{sec} = N\frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}}. \tag{3.26}$$

Надежность системы обычно измеряют *средним временем наработки на отказ* (mean time between failures, MTBF). Как следует из названия, MTBF – это среднее время между отказами системы. Эта величина обратна вероятности сбоя системы за любую заданную секунду:

$$\text{MTBF} = \frac{1}{P(\text{failture})/\text{sec}} = \frac{T_c e^{-\frac{T_c - t_{\text{setup}}}{\tau}}}{NT_0}. \tag{3.27}$$

Выражение **(3.27)** показывает, что MTBF растет экспоненциально с ростом времени ожидания синхронизатора, T_c. Для большинства систем синхронизатор, который ожидает один период тактового сигнала, обеспечивает достаточную величину MTBF. В высокоскоростных системах может понадобиться ожидание на большее количество периодов тактового сигнала.

Пример 3.14 Синхронизатор для входа конечного автомата

Конечный автомат, который управляет работой светофора (**раздел 3.4.1**), принимает асинхронные входные сигналы от датчиков дорожного движения. Предположим, что для обеспечения стабильности входов используются синхронизаторы. В среднем за одну секунду датчик срабатывает 0.2 раза. Триггер в синхронизаторе имеет следующие характеристики: $\tau = 200$ пс, $T_0 = 150$ пс и $t_{setup} = 500$ пс. Каким должен быть период синхронизатора, чтобы среднее время наработки на отказ (MTBF) превышало 1 год?

Решение 1 год $\approx \pi \times 10^7$ секунд.

$$\pi \times 10^7 = \frac{T_c e^{-\frac{T_c - 500\times10^{-12}}{200^{-12}}}}{(0,2)(150\times10^{-12})}. \tag{3.28}$$

Для нахождения искомого периода нужно решить уравнение **(3.27)**, которое не имеет решения в аналитическом виде. Но его достаточно просто решить методом проб и ошибок. В электронной таблице можно попробовать несколько величин T_c и посчитать MTBF, пока не будет найдена величина T_c, которая даст MTBF, близкое к 1 году: $T_c = 3,036$ нс.

3.5.6. Вычисление времени разрешения

Выражение **(3.24)** можно получить, используя базовые знания курсов теории цепей, дифференциальных уравнений и теории вероятностей. Этот раздел можно пропустить, если вы не интересуетесь выводом данного выражения или если вы слабо знакомы с элементарной математикой.

Выход триггера будет метастабильным спустя некоторое время t, если триггер пытается зафиксировать изменяющийся вход (что приводит к возникновению метастабильного состояния) и выход не успевает прийти к корректному уровню в течение этого времени после фронта тактового сигнала. Символически это можно выразить так:

$$P(t_{res} > t) = P(\text{samples changing input}) \times P(\text{unresolved}). \tag{3.29}$$

Оба вероятностных множителя будут рассмотрены отдельно. Как показано на **рис. 3.54**, асинхронный входной сигнал переходит из состояния 0 в состояние 1 в течение некоторого времени t_{switch}. Вероятность того, что вход изменится в течение апертурного времени, равна

$$P(\text{samples changing input}) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{T_C}. \qquad (3.30)$$

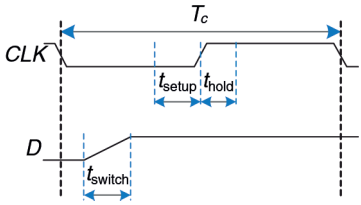


Рис. 3.54 Временная диаграмма входного сигнала

Если триггер уже перешел в метастабильное состояние с вероятностью P(samples changing input), то время, необходимое для разрешения метастабильности, зависит от внутренней структуры схемы. Это время определяет вероятность P(unresolved) – вероятность того, что триггер не успевает перейти в корректное состояние (0 или 1) за время t. В этом разделе будет проанализирована простая модель бистабильного прибора и сделана оценка этой вероятности.

Для построения бистабильного прибора используется запоминающее устройство с положительной обратной связью. На **рис. 3.55 (a)** показана реализация такой обратной связи с использованием двух инверторов; поведение такой схемы является репрезентативным для большинства бистабильных элементов. Пара инверторов ведет себя аналогично буферу. Для построения модели можно считать, что буфер имеет симметричную передаточную характеристику на постоянном токе, которая показана на **рис. 3.55 (b)**, наклон характеристики равен G.

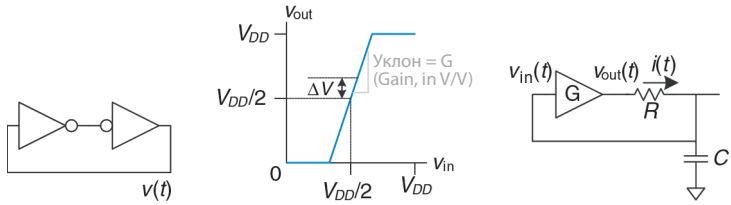


Рис. 3.55 Схемная модель бистабильного устройства

Выходной ток буфера ограничен. Этот факт можно промоделировать его выходным сопротивлением, R. Все реальные схемы имеют также некоторую емкость C, которую нужно перезаряжать при изменении состояния схемы. Процесс зарядки конденсатора через резистор не позволяет буферу переключиться мгновенно, время этого процесса равно RC. Таким образом, полная модель схемы показана на **рис. 3.55 (c)**, где $v_{\text{out}}(t)$ – напряжение, определяющее состояние бистабильной схемы.

Состояние схемы, при котором $\upsilon_{out}(t) = \upsilon_{in}(t) = V_{DD}/2$, является метастабильным; если схема начинает работать с этого состояния, то при отсутствии шума она будет находиться в нем неопределенно долго. Поскольку все напряжения являются непрерывными величинами, то вероятность того, что работа схемы начнется точно в точке метастабильности, исчезающе мала. Но схема может начать работать в нулевой момент времени около точки метастабильности, когда $\upsilon_{out}(0) = V_{DD}/2 + \Delta V$, где ΔV – малое отклонение. В таком случае положительная обратная связь в конце концов приведет $\upsilon_{out}(t)$ к V_{DD}, если $\Delta V > 0$, или к 0, если $\Delta V < 0$. Время, необходимое для достижения V_{DD} или 0, является временем разрешения бистабильного прибора.

Передаточная характеристика буфера по постоянному току нелинейна, но в окрестности точки метастабильности она имеет форму, близкую к линейной. Более точно: если $\upsilon_{in}(t) = V_{DD}/2 + \Delta V/G$, то $\upsilon_{out}(t) = V_{DD}/2 + \Delta V$ для малых ΔV. Ток через резистор равен $i(t) = (\upsilon_{out}(t) - \upsilon_{in}(t))/R$. Конденсатор заряжается со скоростью $d\upsilon_{in}(t)/dt = i(t)/C$. Объединяя эти два выражения, можно найти уравнение для выходного напряжения.

$$\frac{d\upsilon_{out}(t)}{dt} = \frac{(G-1)}{RC}\left[\upsilon_{out} - \frac{V_{DD}}{2}\right]. \tag{3.31}$$

Это линейное дифференциальное уравнение первого порядка. Решая его с начальным условием $\upsilon_{out}(0) = V_{DD}/2 + \Delta V$, можно найти зависимость выходного напряжения от времени:

$$\upsilon_{out}(t) = \frac{V_{DD}}{2} + \Delta V e^{-\frac{(G-1)t}{RC}}. \tag{3.32}$$

На **рис. 3.56** приведены графики $\upsilon_{out}(t)$ для разных начальных точек. Напряжение $\upsilon_{out}(t)$ экспоненциально удаляется от метастабильной точки $V_{DD}/2$, пока не достигнет предела V_{DD} или 0. Выход схемы в конце концов приходит в корректное логическое состояние 0 или 1. Время, необходимое для этого, зависит от отклонения начального напряжения (ΔV) от точки метастабильности $(V_{DD}/2)$.

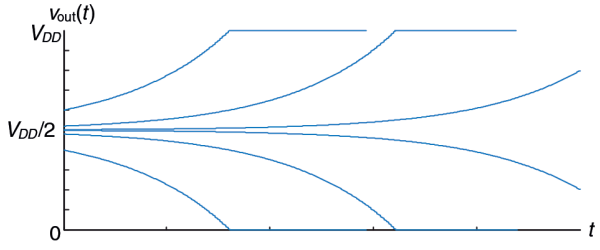


Рис. 3.56 Временная диаграмма процесса схемы в корректное состояние

Если в уравнение **(3.32)** подставить $\upsilon_{out}(t_{res}) = V_{DD}$ или 0, то можно найти время разрешения t_{res}:

$$|\Delta V| e^{-\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2}; \tag{3.33}$$

$$t_{res} = \frac{RC}{G-1} \ln \frac{V_{DD}}{2|\Delta V|}. \tag{3.34}$$

Таким образом, время разрешения возрастает, если бистабильное устройство имеет большое сопротивление или емкость, которые не позволяют выходному напряжению быстро изменяться. Оно уменьшается, если бистабильное устройство имеет большое усиление, G. Время разрешения также логарифмически возрастает при приближении начальных условий схемы к точке метастабильности ($\Delta V \to 0$).

Обозначим τ через $\frac{RC}{G-1}$. Из уравнения **(3.34)** можно получить значение начального отклонения, которое соответствует некоторому заданному времени разрешения t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-\frac{t_{res}}{\tau}}. \tag{3.35}$$

Предположим, что бистабильное устройство пытается зафиксировать входной сигнал во время его изменения. На его вход поступает напряжение $\upsilon_{in}(0)$, которое предполагается равномерно распределенным в интервале от 0 до V_{DD}. Вероятность того, что выход не достигнет корректного значения через время t_{res}, зависит от вероятности того, что начальное отклонение будет достаточно малым. Точнее начальное отклонение υ_{out} должно быть меньше, чем $\Delta V_{res}/G$. Тогда вероятность того, что входной сигнал бистабильного устройства имеет достаточно малое отклонение, равна

$$P(\text{unresolved}) = P\left(\left|\upsilon_{in}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{G V_{DD}}. \tag{3.36}$$

Таким образом, вероятность того, что время разрешения превосходит некоторую заданную величину t, задается следующим выражением:

$$P(t_{res} > t) = \frac{t_{switch} + t_{setup} + t_{hold}}{G T_c} e^{-\frac{t}{\tau}}. \tag{3.37}$$

Обратите внимание на то, что выражения **(3.37)** и **(3.24)** имеют одинаковый вид, если $T_0 = (t_{switch} + t_{setup} + t_{hold})/G$ и $\tau = RC/(G-1)$. Итак, мы вывели выражение **(3.24)** и показали, как величины T_0 и τ зависят от физических свойств бистабильного устройства.

3.6. Параллелизм

Скорость обработки информации системой характеризуется задержкой и пропускной способностью информации передачи информации через нее. Мы определим *токен* (token) как группу входов, которая обрабатывается для того, чтобы получить группу выходов. Это название связано с методом визуализации передачи данных внутри системы путем размещения в схеме токенов или маркеров и их передвижением по схеме вместе с обрабатываемыми данными. *Задержка*, или *латентность* (latency), системы – время, которое необходимо для прохождения одного токена через всю систему с ее входа на выход. *Пропускная способность* (throughput) – количество токенов, которое может быть обработано системой в единицу времени.

Пример 3.15 ПРОПУСКНАЯ СПОСОБНОСТЬ И ЗАДЕРЖКА ПРИ ПРИГОТОВЛЕНИИ ПЕЧЕНЬЯ

Бену нужно быстро подготовиться к вечеринке с молоком и печеньем, посвященной введению в эксплуатацию его светофора. За 5 минут он лепит печенья и укладывает их на противень. В течение 15 минут печенья выпекаются в печи. После окончания выпекания он начинает готовить следующий противень. Какая пропускная способность и задержка выпекания Беном противня печенья?

Решение В этом примере противень является токеном. Задержка равна 1/3 часа на противень. Пропускная способность – 3 противня в час.

Достаточно легко понять, что пропускная способность может быть увеличена путем обработки нескольких токенов в одно и то же время. Это называется *параллелизмом* и используется в двух формах: пространственной и временной. В *пространственном параллелизме* используется несколько копий аппаратных блоков, так что в одно и то же время можно выполнять несколько задач. *Временной параллелизм* предполагает разделение задачи на несколько стадий (или ступеней), как это происходит на сборочном конвейере. Несколько задач могут быть распределены по стадиям. Хотя все задачи должны пройти по всем стадиям, разные задачи в любой заданный момент времени будут находиться на своей стадии, так что несколько задач могут одновременно обрабатываться на разных стадиях. Временной параллелизм часто называется *конвейеризацией*. Пространственный параллелизм нередко называют просто параллелизмом, но мы будем избегать этого названия из-за его неоднозначности.

Пример 3.16 ПАРАЛЛЕЛИЗМ ПРИ ПРИГОТОВЛЕНИИ ПЕЧЕНЬЯ

К Бену Битдидлу на вечеринку придут сотни друзей, и ему нужно печь печенье быстрее. Он собирается использовать пространственный и/или временной параллелизм.

Пространственный параллелизм: Бен просит Алису Хакер помочь ему. У нее есть собственная печь и противень.

Временной параллелизм: Бену дали второй противень. Как только он ставит один противень в печь, он начинает лепить печенья для другого противня, а не ожидает окончания выпекания печенья на первом противне.

Какая будет задержка и пропускная способность при использовании пространственного параллелизма? Временного? При использовании обоих видов параллелизма?

Решение Задержка – это время, необходимое для завершения одной задачи от начала до конца. Во всех случаях задержка равна 1/3 часа. Если в начале у Бена не было печенья, то задержка – это время, необходимое для производства первого противня.

Пропускная способность – это количество противней с печеньем, которое производится за один час. При использовании пространственного параллелизма и Бен, и Алиса делают по одному противню каждые 20 минут. Следовательно, пропускная способность удваивается и составляет 6 противней/час. При использовании временного параллелизма Бен ставит новый противень в печь каждые 15 минут, пропускная способность равна 4 противня/час. Это показано на **рис. 3.57**.

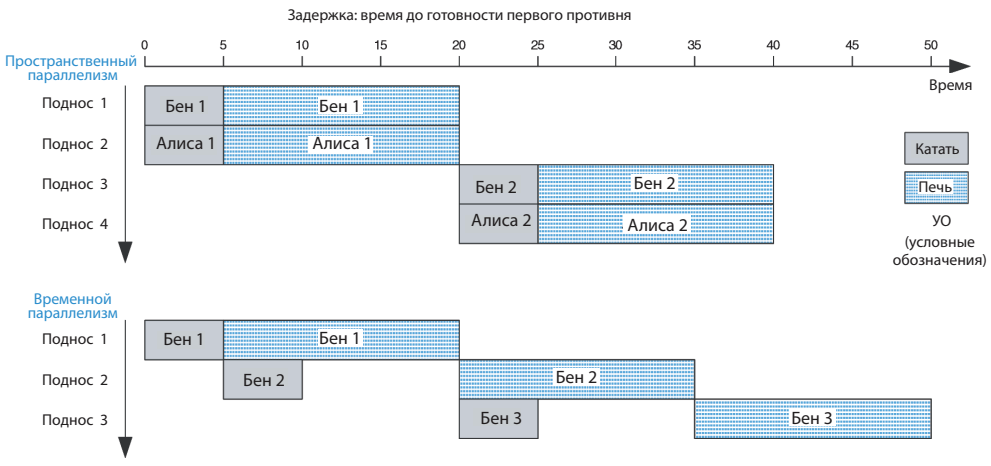


Рис. 3.57 Пространственный и временной параллелизмы при приготовлении печенья

Если Бен и Алиса используют оба метода, они могут выпекать 8 противней/час.

Рассмотрим систему с задержкой L. Если в системе отсутствует параллелизм, то пропускная способность будет $1/L$. В системе с пространственным параллелизмом, которая содержит N копий аппаратных блоков, пропускная способность будет N/L. В системе с временным параллелизмом задача в идеальном случае разбивается на N стадий одинаковой длины. В этом случае пропускная способность будет также равна N/L, причем необходим только один экземпляр аппаратного блока. Но,

как показывает пример приготовления печенья, часто создание N ступеней одной и той же продолжительности обработки невозможно. Если самая длинная ступень имеет задержку L_1, то пропускная способность конвейеризированной системы будет равна $1/L_1$.

Конвейеризация (временной параллелизм) особенно привлекательна, поскольку она увеличивает скорость работы схемы без увеличения аппаратных затрат. Вместо этого регистры, установленные между блоками комбинационной логики, разделяют ее на короткие стадий, которые могут работать на более высокой тактовой частоте. Регистры не позволяют токенам, находящимся на одной стадии, догонять и разрушать токены, которые находятся на следующей стадии обработки.

На **рис. 3.58** приведен пример схемы, в которой отсутствует конвейеризация. Она состоит из четырех блоков логики, которые расположены между двумя регистрами. Критический путь проходит через блоки 2, 3 и 4. Предположим, что регистр имеет задержку распространения на тактовый вход-выход 0,3 нс и время удержания 0,2 нс. Тогда минимальный период тактового сигнала равен $Tc = 0{,}3 + 3 + 2 + 4 + 0{,}2 = 9{,}5$ нс. Схема имеет задержку 9,5 нс и пропускную способность $1/9{,}5$ нс = 105 МГц.

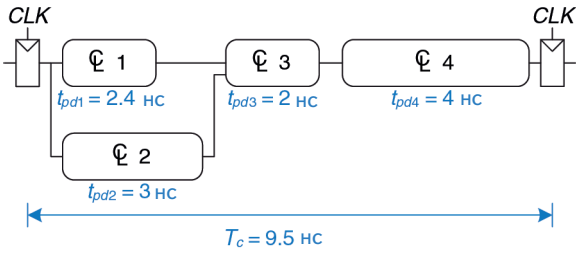


Рис. 3.58 Схема без конвейеризации

На **рис. 3.59** показана эта же самая схема, разделенная с помощью дополнительных регистров между блоками 3 и 4, на 2-стадийный конвейер. Первая стадия имеет минимальный период тактового сигнала $0{,}3 + 3 + 2 + 0{,}2 = 5{,}5$ нс. Минимальный период для второй стадии равен $0{,}3 + 4 + 0{,}2 = 4{,}5$ нс. Тактовый сигнал должен быть достаточно медленным, для того чтобы работали все стадии. Следовательно, $T_c = 5{,}5$ нс. Задержка равна двум периодам тактового сигнала, или 11 нс. Пропускная способность равна $1/5{,}5$ нс = 182 МГц. Этот пример показывает, что в реальных схемах конвейеризация с двумя стадиями почти удваивает пропускную способность и немного увеличивает задержку. Для сравнения: идеальная конвейеризация точно удвоила бы пропускную способность и не ухудшила задержку. Несоответствие возникает потому, что реальную схему невозможно разделить на две абсолютно равные части, и также потому, что конвейерные регистры вносят дополнительные потери на упорядочение.

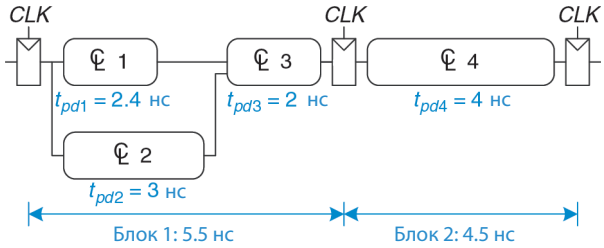


Рис. 3.59 Схема с двухстадийным конвейером

На **рис. 3.60** показан еще один вариант той же схемы, в котором используется трехстадийный конвейер. Обратите внимание, что в схеме необходимо на два регистра больше, они сохраняют результаты блоков 1 и 2 в конце первой стадии конвейера. Время цикла ограничивается теперь третьей стадией и равно 4,5 нс. Задержка равна трем циклам, или 13,5 нс. Пропускная способность равна 1/4,5 нс = 222 МГц. Как и в прошлом варианте схемы, добавление еще одной стадии конвейера улучшает пропускную способность за счет небольшого увеличения задержки.

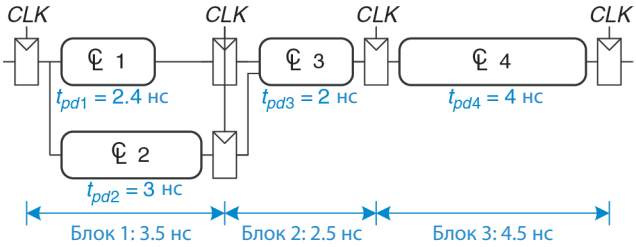


Рис. 3.60 Схема с трехстадийным конвейером

Хотя рассмотренные подходы весьма эффективны, они не могут быть использованы во всех ситуациях. Использование параллелизма ограничивается *взаимозависимостями* (dependencies) реальных задач. Если текущая задача зависит от результатов предыдущей задачи, а не только от своих предыдущих шагов, то выполнение задачи не может быть начато до завершения предыдущей задачи. Например, если Бен Битдидл хочет проверить, достаточно ли вкусны печенья из первого противня перед приготовлением второго, то имеется взаимозависимость, которая препятствует использованию конвейера или параллелизма. Параллелизм – один из самых важных методов разработки высокопроизводительных цифровых систем. Конвейеризация будет далее обсуждаться в **главе 7**, там же будут показаны примеры обработки взаимозависимостей.

3.7. Заключение

Эта глава посвящена рассмотрению методов анализа и разработки последовательностных схем. В отличие от комбинационных схем, выходные сигналы которых зависят только от текущих состояний входных сигналов, выходные сигналы последовательностных схем зависят как от текущих, так и от предыдущих состояний входных сигналов. Другими словами, последовательностная схема помнит информацию о входных сигналах в предыдущие моменты времени. Эта память называется состоянием схемы.

Любой, кто сможет изобрести схемы, выходы которой зависят от будущих входов, станет фантастически богатым!

Последовательностные схемы могут быть сложны для анализа, и их легко неправильно спроектировать, поэтому мы ограничимся использованием небольшого количества тщательно разработанных аппаратных блоков. Наиболее важным элементом для наших целей является триггер, который принимает тактовый сигнал и входной сигнал D и формирует выходной сигнал Q. По переднему фронту тактового импульса триггер копирует вход D на выход Q, в противном случае он сохраняет старое состояние Q. Группа триггеров с общим тактовым сигналом называется регистром. На триггеры могут также поступать управляющие сигналы сброса или разрешения.

Хотя существует множество видов последовательностных схем, мы ограничимся использованием синхронных последовательностных схем, поскольку их просто разрабатывать. Синхронные последовательностные схемы состоят из блоков комбинационной логики, разделенных тактируемыми регистрами. Состояние схемы сохраняется в регистрах и обновляется только по фронтам тактового сигнала.

Один из эффективных подходов к разработке последовательностных схем основывается на использовании конечных автоматов. Для разработки конечного автомата сначала следует определить его входы и выходы, потом сделать эскиз диаграммы переходов с указанием состояний и условий переходов между ними. Затем для всех состояний автомата нужно выбрать способ кодирования состояний и на основе диаграммы создать таблицу переходов между состояниями и таблицу выходов, которые показывают следующее состояние и выходной сигнал при заданном текущем состоянии и входном сигнале. По этим таблицам разрабатывают комбинационную логическую схему, которая определяет следующее состояние и выходной сигнал, и создается эскиз схемы.

Синхронные последовательностные схемы характеризуются временной спецификацией, которая включает в себя задержки распространения и реакции тракта тактовый вход-выход, t_{pcq} и t_{ccq}, а также временами предустановки и удержания, t_{setup} и t_{hold}. Для корректной работы схем их входы должны быть стабильными в течение апертурного времени, которое состоит из времени предустановки перед передним фронтом такто-

вого импульса и времени удержания после него. Минимальный период T_c тактового сигнала системы равен сумме задержек распространения комбинационной логики, t_{pd}, и задержек $t_{pcq} + t_{\text{setup}}$ в регистрах. Для корректной работы схемы задержка реакции регистров и комбинационной логики должна быть больше, чем t_{hold}. Несмотря на распространенное заблуждение, время удержания не влияет на величину минимального периода тактового сигнала.

Общая производительность системы измеряется задержкой и пропускной способностью. Задержка – это время, необходимое для прохождения одного токена с входа системы на ее выход. Пропускная способность – количество токенов, которое система может обработать в единицу времени. Параллелизм увеличивает пропускную способность системы.

Упражнения

Упражнение 3.1 Временные диаграммы входных сигналов RS-защелки показаны на **рис. 3.61**. Нарисуйте временную диаграмму значений выхода Q.

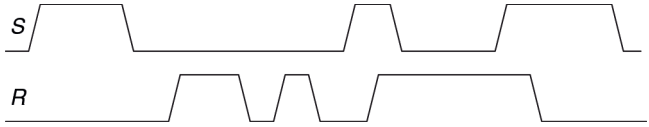


Рис. 3.61 Временные диаграммы входов RS-защелки для упражнения 3.1

Упражнение 3.2 Временные диаграммы входных сигналов RS-защелки показаны на **рис. 3.62**. Нарисуйте временную диаграмму значений выхода Q.

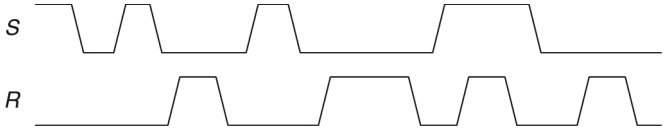


Рис. 3.62 Временные диаграммы входов RS-защелки для упражнения 3.2

Упражнение 3.3 Временные диаграммы входных сигналов D-защелки показаны на **рис. 3.63**. Нарисуйте временную диаграмму значений выхода Q.

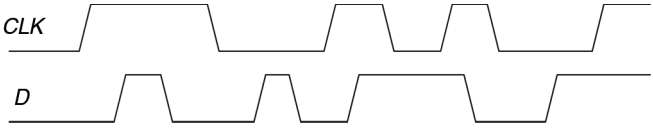


Рис. 3.63 Временные диаграммы входов D-защелки или D-триггера для упражнений 3.3 и 3.5

Упражнение 3.4 Временные диаграммы входных сигналов D-защелки показаны на **рис. 3.64**. Нарисуйте временную диаграмму значений выхода Q.

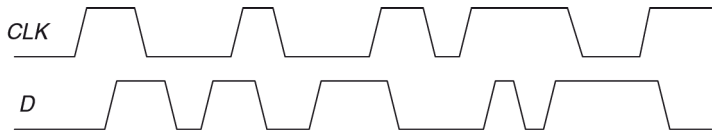


Рис. 3.64 Временные диаграммы входов D-защелки или D-триггера для упражнений 3.4 и 3.6

Упражнение 3.5 На **рис. 3.63** показаны временные диаграммы входов D-триггера (синхронизируемого фронтом). Нарисуйте временную диаграмму значений выхода Q.

Упражнение 3.6 На **рис. 3.64** показаны временные диаграммы входов D-триггера (синхронизируемого фронтом). Нарисуйте временную диаграмму значений выхода Q.

Упражнение 3.7 Является ли схема, изображенная на **рис. 3.65**, комбинационной или последовательностной? Объясните взаимосвязь входов с выходами. Как называется такая схема?

Упражнение 3.8 Является ли схема, изображенная на **рис. 3.66**, комбинационной или последовательностной? Объясните взаимосвязь входов с выходами. Как называется такая схема?

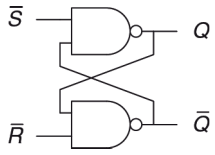


Рис. 3.65 Исследуемая схема

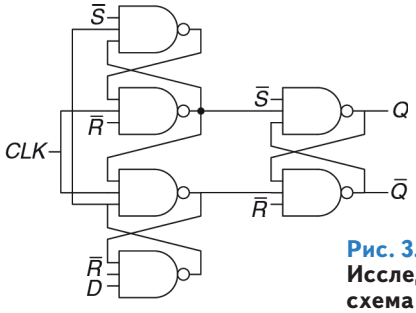


Рис. 3.66 Исследуемая схема

Упражнение 3.9 *T-триггер* (от англ. *toggle* – переключать) имеет один вход *CLK* и один выход Q. По каждому фронту тактового сигнала значение на выходе триггера изменяется на противоположное. Нарисуйте схему T-триггера, используя D-триггер и инвертор.

Упражнение 3.10 На вход JK-триггера поступают тактовый сигнал *CLK* и входные данные J и K. Триггер синхронизируется по фронту тактового сигнала. В случае если J и K равны нулю, то на выходе Q сохраняется старое значение. Если $J = 1$, $K = 0$, то Q устанавливается в 1. Если $J = 0$, $K = 1$, то Q сбрасывается в 0. Если $J = 1$, $K = 1$, то Q принимает противоположное значение.

a) Постройте JK-триггер, используя D-триггер и комбинационную логику.
b) Постройте D-триггер, используя JK-триггер и комбинационную логику.

c) Постройте T-триггер (**упражнение 3.9**), используя JK-триггер.

Упражнение 3.11 Схема, изображенная на **рис. 3.67**, называется *C-элементом Мюллера*. Объясните взаимосвязь входов с выходами.

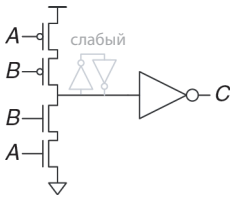


Рис. 3.67 C-элемент Мюллера

Упражнение 3.12 Разработайте D-защелку с асинхронным сбросом, используя логические элементы.

Упражнение 3.13 Разработайте D-триггер с асинхронным сбросом, используя логические элементы.

Упражнение 3.14 Разработайте синхронно устанавливаемый D-триггер, используя логические элементы.

Упражнение 3.15 Разработайте асинхронно устанавливаемый D-триггер, используя логические элементы.

Упражнение 3.16 Кольцевой генератор состоит из N инверторов, замкнутых в кольцо. У каждого инвертора есть минимальная t_{cd} и максимальная t_{pd} задержки. Определите диапазон частот, в котором может работать кольцевой генератор, при условии что N нечетное.

Упражнение 3.17 Почему число N из **упражнения 3.16** должно быть нечетным?

Упражнение 3.18 Какие из схем на **рис. 3.68** являются синхронными и последовательностными? Дайте развернутый ответ.

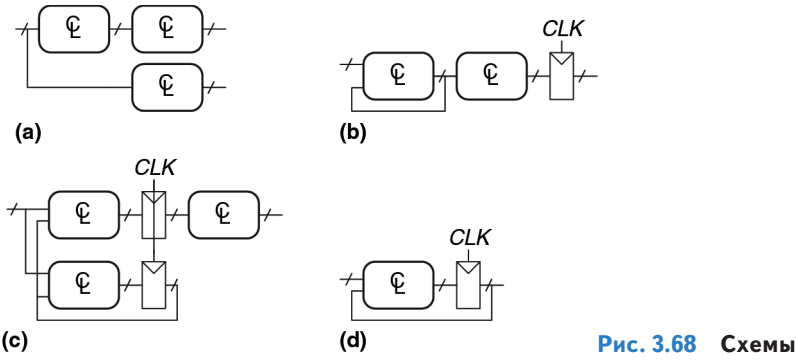


Рис. 3.68 Схемы

Упражнение 3.19 Вы разрабатываете контроллер лифта для 25-этажного здания. У контроллера есть два входа: ВВЕРХ и ВНИЗ. Выходными данными является номер этажа, на котором находится лифт. 13-й этаж отсутствует. Чему равно минимальное количество битов для хранения состояния в контроллере?

Упражнение 3.20 Вы разрабатываете конечный автомат для отслеживания настроения четырех студентов, работающих в лаборатории по разработке цифровых схем. У студентов может быть следующее настроение: СЧАСТЛИВЫЙ (если схема работает), ГРУСТНЫЙ (если схема сгорела), ЗАНЯТЫЙ (работает над схемой), ЗАГРУЖЕННЫЙ (думает над схемой), СПЯЩИЙ (спит на рабочем месте). Сколько состояний будет у вашего автомата? Какое минимальное количество битов состояний необходимо для кодирования состояния автомата?

Упражнение 3.21 Как бы вы разделили конечный автомат из **упражнения 3.20** на несколько менее сложных автоматов? Сколько состояний было бы у каждого такого простого автомата? Какое минимальное количество битов необходимо для такого модульного проекта?

Упражнение 3.22 Опишите словами, что делает автомат на **рис. 3.69**. Заполните таблицу переходов и таблицу выходов, используя двоичное кодирование. Составьте логические выражения для следующего состояния и для выхода и разработайте схему этого конечного автомата.

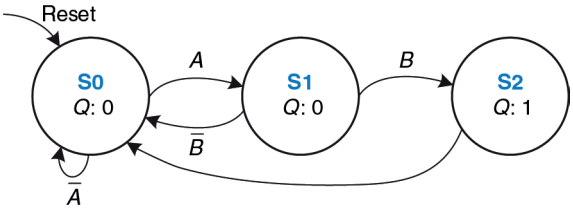


Рис. 3.69 Диаграмма переходов

Упражнение 3.23 Опишите словами, что делает автомат на **рис. 3.70**. Заполните таблицу переходов и таблицу выходов, используя двоичное кодирование. Составьте логические выражения для следующего состояния и для выхода и разработайте схему этого конечного автомата.

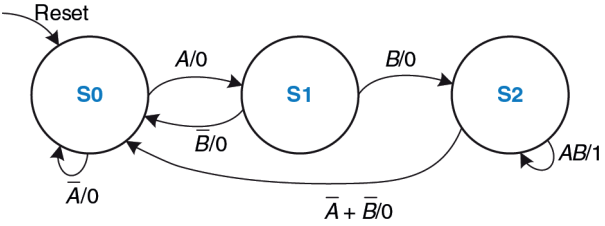


Рис. 3.70 Диаграмма переходов

Упражнение 3.24 На пересечении Академической и Беговой улиц все еще случаются происшествия. Футболисты выбегают на перекресток, как только на их светофоре загорается зеленый свет, и сталкиваются с зазевавшимися ботаниками. Последние выходят на перекресток все еще на зеленый свет. Усовершенствуйте светофор из **раздела 3.4.1** так, чтобы на обеих улицах горел красный свет в течение 5 секунд до того, как какой-либо из светофоров станет зеленым. Разработайте диаграмму переходов автомата Мура, кодирование состояний, таблицу переходов, таблицу выходов, выражения для определения выходов и для следующего состояния и схему конечного автомата.

Упражнение 3.25 У улитки Алисы из **раздела 3.4.3** есть дочка, которая перемещается под управлением автомата Мили. Улитка-дочка улыбается, когда она проходит последовательность 1101 или 1110. Нарисуйте диаграмму переходов для этой веселой улитки, используя как можно меньше состояний. Выберите кодирование состояний и составьте общую таблицу переходов и выходов. Составьте выражения для выхода и для следующего состояния и нарисуйте схему автомата.

Упражнение 3.26 Вас уговорили разработать автомат с прохладительными напитками для офиса. Расходы на напитки частично покрывает профсоюз, поэтому они стоят всего по 5 рублей. Автомат принимает монеты номиналом в 1, 2 и 5 рублей. Как только покупатель внесет необходимую сумму, автомат выдаст напиток и сдачу. Разработайте конечный автомат для автомата с прохладительными напитками. Входами автомата являются 1, 2 и 5 рублей (монета, вставленная в данный момент в монетоприемник). Предположим, что по каждому тактовому сигналу вставляется только одна монета. Автомат имеет выходы: налить газировку, вернуть 1 рубль, вернуть 2 рубля, вернуть 2 по 2 рубля. Как только в автомате набирается 5 рублей (или больше), он выставляет сигнал «НАЛИТЬ ГАЗИРОВКУ», а также сигналы возврата соответствующей сдачи. Затем автомат должен быть готов опять принимать монеты.

Упражнение 3.27 У кода Грея есть полезное свойство: коды соседних чисел отличаются друг от друга только в одном разряде. В **табл. 3.23** представлен 3-разрядный код Грея, представляющий числовую последовательность от 0 до 7. Разработайте 3-разрядный автомат счетчика в коде Грея по модулю 8. У автомата нет входов, но есть 3 выхода. (Счетчик по модулю N считает от 0 до $N - 1$, затем цикл повторяется. Например, в часах используется счетчик по модулю 60, для того чтобы считать минуты и секунды от 0 до 59.) После сброса на счетчике должно быть 000. По каждому переднему фронту тактового сигнала счетчик должен переходить к следующему коду Грея. По достижении кода 100 счетчик должен опять перейти к коду 000.

Таблица 3.23 3-разрядный код Грея

Числа	Код Грея		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Упражнение 3.28 Усовершенствуйте свой автомат счетчика в коде Грея из **упражнения 3.27** так, чтобы он мог считать как вверх, так и вниз. У счетчика появится вход ВВЕРХ. Если ВВЕРХ = 1, то счетчик будет переходить к следующему коду, а если ВВЕРХ = 0 – то к предыдущему.

Упражнение 3.29 Ваша компания, Детекторама, хочет разработать конечный автомат с двумя входами A и B и одним выходом Z. Выход в n-м цикле, Z_n, является результатом логического И или логического ИЛИ текущего A_n и предыдущего A_{n-1} значений на входе, в зависимости от сигнала B_n.

$$Z_n = A_n A_{n-1}, \quad \text{если } B_n = 0;$$
$$Z_n = A_n + A_{n-1}, \quad \text{если } B_n = 1.$$

a) Нарисуйте временную диаграмму для Z по данным диаграммам A и B, изображенным на **рис. 3.71**.
b) Этот автомат является автоматом Мура или автоматом Мили?
c) Разработайте конечный автомат. Составьте диаграмму переходов, закодированную таблицу переходов, выражения для выходов и следующего состояния и нарисуйте схему.

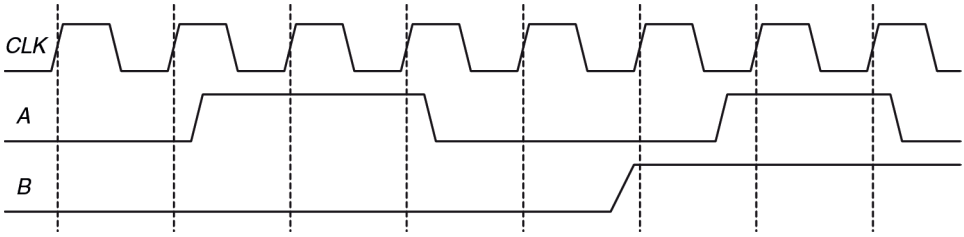


Рис. 3.71 Входные временные диаграммы конечного автомата

Упражнение 3.30 Разработайте конечный автомат с одним входом A и двумя выходами X и Y. На выходе X должна появиться 1, если 1 поступали на вход как минимум 3 цикла (необязательно подряд), а на Y должна появиться 1, если $X = 1$ как минимум 2 цикла подряд. Составьте диаграмму переходов, закодированную таблицу переходов, выражения для выходов и следующего состояния и нарисуйте схему.

Упражнение 3.31 Проанализируйте конечный автомат, показанный на **рис. 3.72**. Составьте таблицу переходов и выходов, а также диаграмму состояний. Опишите словами, что делает этот автомат.

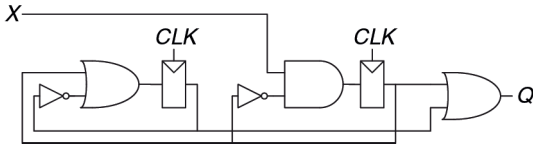


Рис. 3.72 Схема конечного автомата

Упражнение 3.32. Повторите **упражнение 3.31** со схемой, показанной на **рис. 3.73**. Входы регистров s и r отвечают за установку (Set) и сброс (Reset) соответственно.

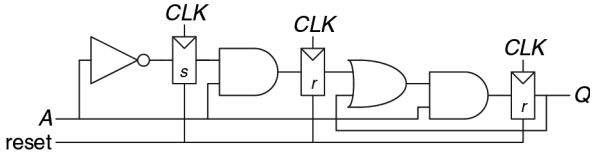


Рис. 3.73 Схема конечного автомата

Упражнение 3.33 Бен Битдидл разработал схему вычисления функции XOR с четырьмя входами и регистрами на входе и выходе (**рис. 3.74**). Каждый двухвходовый элемент XOR имеет задержку распространения 100 пс и задержку реакции 55 пс. Время предустановки триггеров равно 60 пс, время удержания – 20 пс, максимальная задержка тактовый сигнал – выход равна 70 пс, минимальная задержка – 50 пс.

a) Чему будет равна максимальная рабочая частота схемы при отсутствии расфазировки тактовых импульсов?
b) Какая расфазировка тактовых импульсов допустима, если схема должна работать на частоте 2 ГГц?
c) Какая расфазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?
d) Алиса Хакер утверждает, что она может изменить комбинационную логическую схему с целью повышения ее скорости и устойчивости к расфазировке тактовых импульсов. В ее улучшенной схеме также используется три двухвходовых элемента XOR, но они по-другому соединены между собой. Какую схему она разработала? Какая у нее будет максимальная частота без расфазировки тактовых импульсов? Какая расфазировка тактовых импульсов допустима до возникновения нарушений ограничений времени удержания?

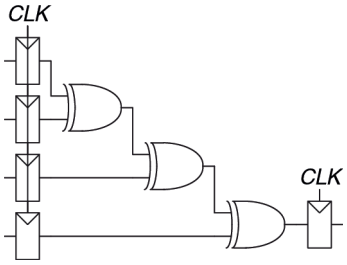


Рис. 3.74 Схема вычисления функции XOR с регистрами на входе и выходе

Упражнение 3.34 В рамках проектирования сверхбыстродействующего двухразрядного процессора RePentium вам поручена разработка сумматора. Как показано на **рис. 3.75**, сумматор состоит из двух полных сумматоров, выход переноса первого сумматора подсоединен ко входу переноса второго. На входе и выходе сумматора находятся регистры, сумматор должен выполнить сложение за один период тактового сигнала. Задержки распространения полных сумматоров равны: в тракте вход C_{in} – выходы C_{out} и S_{um} (S) – 20 пс, по тракту входы A и B – выход C_{out} – 25 пс, в тракте входы A и B – выход S – 30 пс. Полные сумматоры имеют задержки реакции: в тракте вход C_{in} – любой выход – 15 пс, в тракте входы A и B – любой выход – 22 пс. Время предустановки триггеров равно 30 пс, время удержания – 10 пс, задержки тракта тактовый сигнал – выход: распространения 10 пс, реакции 21 пс.

a) Чему будет равна максимальная рабочая частота схемы при отсутствии расфазировки тактовых импульсов?
b) Какая расфазировка тактовых импульсов допустима, если схема должна работать на частоте 8 ГГц?

c) Какая расфазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?

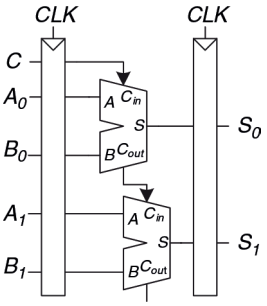


Рис. 3.75 Схема двухразрядного сумматора

Упражнение 3.35 В ПЛИС (*field programmable gate array*, *FPGA*) для создания комбинационных логических схем используются конфигурируемые логические блоки (*configurable logic blocks*, *CLBs*), а не логические элементы.
В матрицах Xilinx Spartan 3 задержки распространения и реакции каждого CLB равны 0,61 и 0,30 нс соответсвенно. Они также содержат триггеры, задержки распространения и реакции которых равны 0,72 и 0,50 нс, а времена предустановки и удержания – 0,53 и 0 нс соответственно.

a) Если вы проектируете систему, которая должна работать на частоте 40 МГц, сколько последовательно соединенных CLB можно разместить между двумя триггерами? При ответе можно считать, что расфазировка тактовых импульсов и задержка в соединениях между CLB отсутствует.
b) Предположим, что все пути между триггерами проходят через, по крайней мере, один CLB. Какая рафазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?

Упражнение 3.36 Для построения синхронизатора используются два триггера с t_{setup} = 50 пс, T_0 = 20 пс, τ = 30 пс. Асинхронный вход изменяется 108 раз за секунду. Чему равен минимальный период синхронизатора, при котором среднее время между отказами (MTBF) достигнет 100 лет?

Упражнение 3.37 Вам необходимо построить синхронизатор, который принимает асинхронные входные сигналы. При этом среднее время между отказами (MTBF) должно быть не менее 50 лет. Тактовая частота системы равна 1 ГГц, триггеры имеют следующие параметры: τ = 100 пс, T_0 = 110 пс, t_{setup} = 70 пс. На вход синхронизатора каждые 2 секунды поступает новый асинхронный сигнал. Чему равна вероятность отказа, которая соответствует заданному среднему времени между отказами (MTBF)? Сколько периодов тактового сигнала следует выждать перед считыванием зафиксированного входного сигнала для достижения этой вероятности?

Упражнение 3.38 Вы столкнулись со своим напарником по лабораторным работам в коридоре, когда он шел навстречу вам. Оба вы отступили в одну сторону и все еще находитесь на пути друг друга. Затем вы оба отступили в другую сторону и продолжаете мешать друг другу пройти. Далее вы оба решили чуть подождать, в надежде, что встречный отступит в сторону и вы разойдетесь. Вы можете промоделировать эту ситуацию как метастабильную и применить к ней ту же

теорию, которая была описана для синхронизаторов и триггеров. Предположим, вы создаете математическую модель своего поведения и поведения своего напарника. Состояние, в котором вы мешаете проходу друг друга, можно трактовать как метастабильное. Вероятность того, что вы остаетесь в этом состоянии после t секунд, равна $e^{-t/\tau}$, величина τ описывает скорость вашей реакции, сегодня из-за недосыпания ваш разум затуманен и $\tau = 20$ с.

a) Через какое время с вероятностью 99 % метастабильность будет разрешена (то есть вы сможете обойти друг друга)?
b) Вы не только не выспались, но и сильно проголодались. Ситуация крайне серьезная, вы умрете от голода, если не попадете в кафетерий через 3 минуты. Какая вероятность того, что ваш напарник по лабораторным работам должен будет доставить вас в морг?

Упражнение 3.39 Вы построили синхронизатор с использованием триггеров с $T_0 = 20$ пс и $\tau = 30$ пс. Ваш начальник поручил вам увеличить среднее время между отказами (MTBF) в 10 раз. Насколько вам нужно увеличить период тактового сигнала?

Упражнение 3.40 Бен Битдидл изобрел новый улучшенный синхронизатор, который по его заявлениям подавляет метастабильность за один период. Схема улучшенного синхронизатора показана на **рис. 3.76**. Бен утверждает, что схема в блоке M представляет собой аналоговый «детектор метастабильности», который выдает сигнал высокого логического уровня, если напряжение на его входе попадает в запретную зону между V_{IL} и V_{IH}. Детектор метастабильности проверяет, не появился ли на выходе D2 первого триггера метастабильный сигнал. Если он действительно появился, то «детектор метастабильности» асинхронно сбрасывает триггер, и на его выходе появляется корректный логический сигнал 0. Второй триггер фиксирует сигнал D2, и на его выходе Q всегда будет корректный логический уровень. Алиса Хакер говорит Бену, что схема не будет работать как заявлено, поскольку устранение метастабильности так же невозможно, как и построение вечного двигателя. Кто из них прав? Покажите, где ошибка Бена или почему Алиса ошибается.

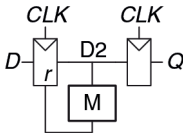


Рис. 3.76 Новый улучшенный синхронизатор

Вопросы для собеседования

В этом разделе представлены типовые вопросы, которые могут быть заданы соискателям при поиске работы в области разработки цифровых систем.

Вопрос 3.1 Нарисуйте диаграмму конечного автомата, который детектирует поступление на вход последовательности 01010.

Вопрос 3.2 Разработайте конечный автомат, который принимает последовательность битов (один бит за раз) и выполняет над ними операцию преобразования в дополнительный код. Он имеет два входа, *Start* и *A*, и один выход *Q*. Двоичное число произвольной длины подается на вход *A*, начиная с младшего разряда. Соответствующий выходной бит появляется на том же цикле на выходе *Q*. Вход *Start* устанавливается на один цикл для инициализации конечного автомата перед поступлением младшего бита.

Вопрос 3.3 Чем отличается защелка от триггера? Когда каждый из них следует использовать?

Вопрос 3.4 Разработайте конечный автомат, который выполняет функцию пятиразрядного счетчика.

Вопрос 3.5 Разработайте схему детектирования переднего фронта сигнала. Ее выход должен принимать значение 1 в течение одного периода после перехода входного сигнала из состояния 0 в 1.

Вопрос 3.6 Опишите концепцию конвейеризации и методы ее использования.

Вопрос 3.7 Опишите ситуацию, когда время удержания триггера может быть отрицательным.

Вопрос 3.8 Разработайте схему, которая принимает сигнал *A* (**рис. 3.77**) и формирует на выходе сигнал *B*.

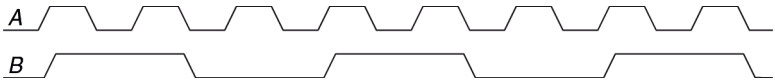
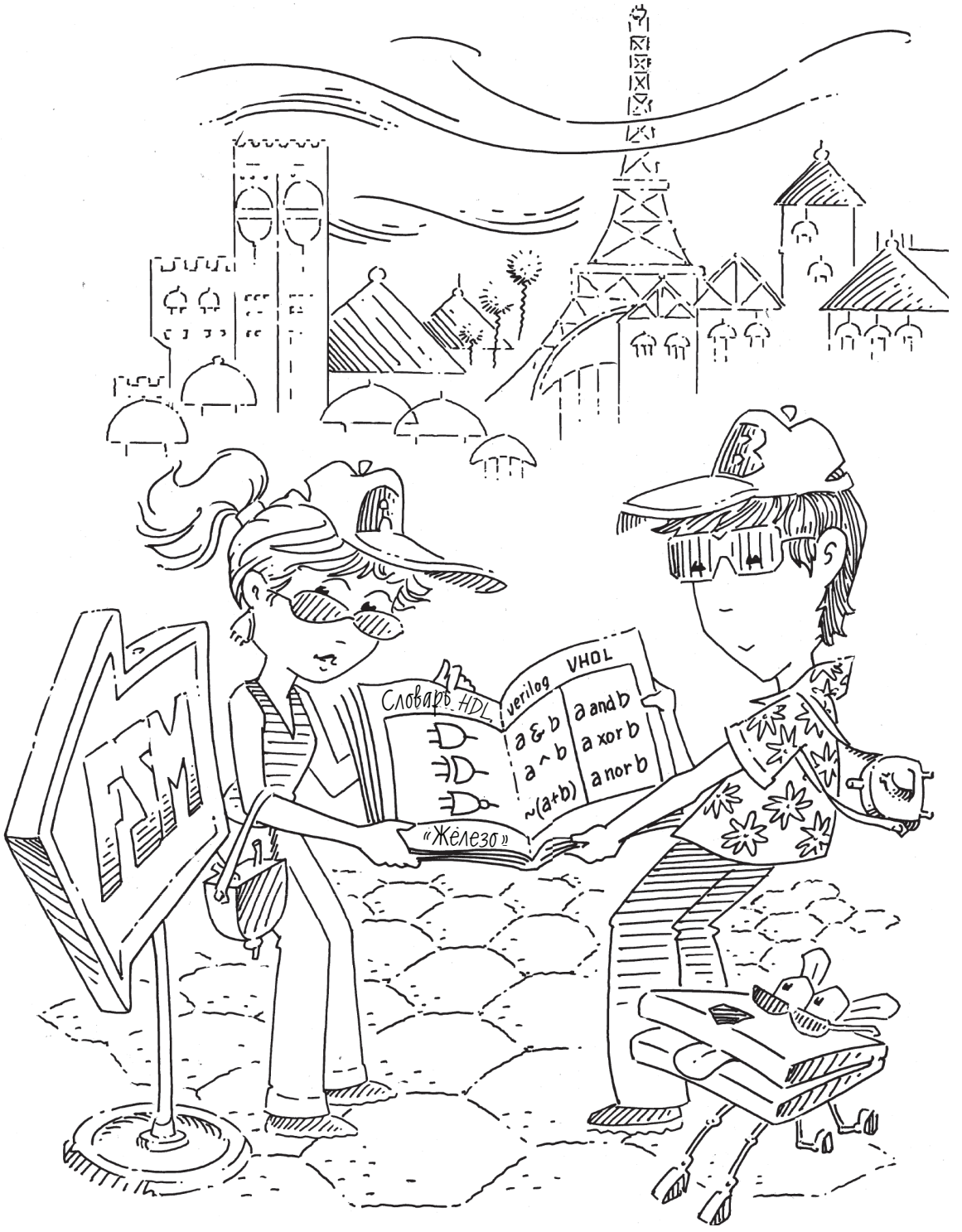


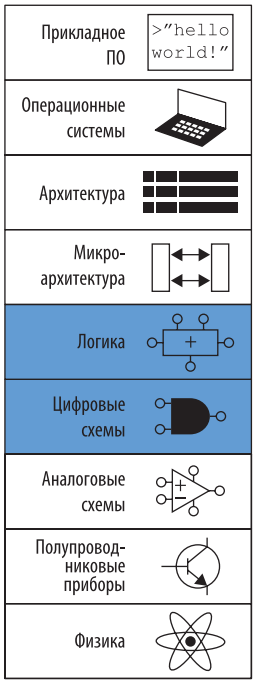
Рис. 3.77 Пример вейвформы для вопроса 3.8

Вопрос 3.9 Рассмотрим блок комбинационной логики между двумя регистрами. Опишите временные ограничения, которым такой блок должен удовлетворять. Если поставить буфер на тактовом входе второго триггера, станут ограничения времени предустановки мягче или жестче?


Словарь HDL
verilog
VHDL
a & b
a and b
a ^ b
a xor b
~(a+b)
a nor b
«Железо»

Глава 4

Языки описания аппаратуры



4.1. Введение

До сих пор мы рассматривали разработку комбинационных и последовательностных цифровых схем на уровне схемотехники. Процесс поиска наилучшего набора логических элементов для выполнения заданной логической функции трудоемок и может приводить к ошибкам, так как требует упрощения логических таблиц или выражений и перевода конечных автоматов в представление на уровне логических элементов вручную. В 1990-е годы разработчики обнаружили, что их производительность труда резко возрастала, если они работали на более высоком уровне абстракции, определяя только логическую функцию и предоставляя создание оптимизированных логических схем *системе автоматического проектирования* (САПР). Два основных *языка описания аппаратуры* (Hardware Description Language, HDL) – SystemVerilog и VHDL.

SystemVerilog и VHDL построены на похожих принципах, но их синтаксис весьма различается. Их обсуждение в этой главе разделено на две колонки для сравнения, где SystemVerilog будет слева, а VHDL – справа. При первом чтении сосредоточьтесь на одном из языков. Как только вы разберетесь с одним, при необходимости вы сможете быстро усвоить другой. В последующих главах аппаратные блоки представлены в виде схем и в форме HDL-моделей и в схематическом виде, и в форме HDL-модели. Если вы решите пропустить эту главу и не изучать языки описания цифровой аппаратуры, вы тем не менее сможете постичь принципы архитектуры микропроцессоров на уровне схем. При этом следует понимать, что подавляющее большинство коммерческих систем сейчас строятся с использованием языков описания цифровой аппаратуры, а не на уровне схемотехники. Если вы когда-либо в вашей карьере собираетесь заниматься разработкой цифровых схем, мы настоятельно рекомендуем вам выучить один из языков описания аппаратуры.

4.1.1. Модули

Блок цифровой аппаратуры, имеющий входы и выходы, называется *модулем*. Логический элемент И, мультиплексор и схема приоритетов являются примерами модулей цифровой аппаратуры. Есть два общепринятых типа описания функциональности модуля – поведенческий и структурный. Поведенческая модель описывает, что модуль делает. Структурная модель описывает то, как построен модуль из более простых элементов, с применением принципа иерархии. Код на SystemVerilog и VHDL из **HDL-примера 4.1** показывает поведенческое описание модуля, который реализует логическую функцию из **примера 2.6**. На обоих языках модуль назван `sillyfunction` и имеет 3 входа, `a`, `b` и `c`, и один выход `y`, и, как и следовало ожидать, следует принципу модульности. Он имеет полностью определенный интерфейс, состоящий из его входов и выходов, и выполняет определенную функцию. Конкретный способ, которым модуль был описан, неважен для тех, кто будет использовать модуль в будущем, поскольку модуль выполняет свою функцию.

4.1.2. Происхождение языков SystemVerilog и VHDL

Примерно в половине вузов, где преподают цифровую схемотехнику, изучают VHDL, а в оставшейся половине – Verilog. В промышленности склоняются к SystemVerilog, но многие компании все еще используют VHDL, поэтому многим разработчикам нужно владеть обоими языками. По сравнению с SystemVerilog, VHDL более многословный и громоздкий, чем можно было бы ожидать от языка, разработанного комитетом[1].

[1] «Верблюд – это лошадь, разработанная комитетом» – американская шутка. – *Прим. перев.*

HDL-пример 4.1 КОМБИНАЦИОННАЯ ЛОГИКА

SystemVerilog

```
module sillyfunction(input logic a, b,
c, output logic y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b & c;

endmodule
```

Модуль на SystemVerilog начинается с имени модуля и списка входов и выходов. Оператор `assign` описывает комбинационную логику. Тильда (~) означает НЕ, амперсанд (&) – И, а вертикальная черта (|) – ИЛИ.

Сигналы типа `logic`, как входы и выходы в примере, – логические переменные, принимающие значения 0 или 1. Они также могут принимать высокоимпедансное и неопределенное значения – это обсуждается в **разделе 4.2.8**.

Тип `logic` появился в SystemVerilog. Он введен для замены типа `reg`, бывшего постоянным источником затруднений в Verilog. Тип `logic` стоит использовать везде, кроме описания сигналов с несколькими источниками. Такие сигналы называются *цепями* (nets) и будут объяснены в **разделе 4.7**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);

end;
architecture synth of sillyfunction is
begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);

end;
```

Код на VHDL состоит из трех частей: объявления используемых библиотек и внешних объектов (`library`, `use`), объявления интерфейса объекта (`entity`) и его внутренней структуры (`architecture`).

Конструкция для объявления используемых внешних объектов будет рассматриваться в **разделе 4.7.2**. В объявлении интерфейса указывается имя модуля и перечисляются его входы и выходы. Блок `architecture` определяет, что модуль делает.

У сигналов в VHDL, в том числе входов и выходов, должен быть указан тип. Цифровые сигналы стоит объявлять как `STD_LOGIC`. Сигналы этого типа принимают значения `'0'` или `'1'`, а также высокоимпедансное и неопределенное значения, которые будут описаны в **разделе 4.2.8**. Тип `STD_LOGIC` определен в библиотеке `IEEE.STD_LOGIC_1164`, поэтому библиотеку объявлять обязательно.

VHDL не определяет соотношение приоритетов операций И и ИЛИ, поэтому при записи логических выражений нужно всегда использовать скобки.

На обоих языках можно полностью описать любую электронную систему, но у каждого языка есть свои особенности. Лучше использовать язык, который уже распространен в вашей организации, или тот, которого требуют ваши клиенты. Большинство САПР сейчас позволяют смешивать языки, поэтому разные модули могут быть разработаны на разных языках.

SystemVerilog

Verilog был разработан компанией Gateway Design Automation в 1984 году как проприетарный язык для моделирования логических схем. В 1989 году Gateway приобрела компания Cadence, и Verilog стал открытым стандартом в 1990 году под управлением сообщества Open Verilog International. Язык стал стандартом IEEE в 1995 году. В 2005 году язык был расширен для устранения противоречий в языке и лучшей поддержки моделирования и верификации систем. Эти расширения были объединены в единый стандарт, который сейчас называется SystemVerilog (стандарт IEEE 1800-2009). Файлы языка SystemVerilog обычно имеют расширение `.sv`.

VHDL

Аббревиатура VHDL расшифровывается как VHSIC Hardware Description Language. VHSIC, в свою очередь, происходит от сокращения Very High Speed Integrated Circuits – названия программы министерства обороны США. Разработка VHDL была начата в 1981 году министерством обороны для описания структуры и функциональности электронных схем. За основу для разработки был взят язык программирования ADA. Изначальной целью языка была документация, но затем он был быстро адаптирован для моделирования и синтеза. IEEE стандартизировал его в 1987 году, и после этого язык обновлялся несколько раз. Эта глава основана на редакции VHDL 2008 года (стандарт IEEE 1076-2008), которая во многих аспектах упорядочивает язык. На момент подготовки этой книги все еще не все функции стандарта VHDL 2008 года поддерживаются в САПР; эта глава использует только те функции, которые поддерживаются в Synplicity, Altera Quartus и Modelsim. Файл языка VHDL имеет расширение `.vhd`.

4.1.3. Моделирование и синтез

Две основные цели HDL – моделирование и синтез цифровых схем. Во время моделирования на входы модуля подаются некоторые воздействия и проверяются выходы, чтобы убедиться, что модуль функционирует корректно. Во время синтеза текстовое описание модуля преобразуется в логические элементы.

Моделирование

Люди регулярно совершают ошибки. Ошибки в цифровой аппаратуре называют багами. Ясно, что устранение багов в цифровой системе очень важно, особенно когда от правильной работы аппаратуры зависят чьи-то жизни. Тестирование системы в лаборатории весьма трудоемко. Исследовать причины ошибок в лаборатории может быть очень сложно, так как наблюдать можно только сигналы, подключенные к контактам чипа, а то, что происходит внутри чипа, напрямую наблюдать невозможно. Исправление ошибок уже после того, как система была выпущена, может быть очень дорого. Например, исправление одной ошибки в новейших интегральных микросхемах стоит больше миллиона долларов и занимает несколько месяцев. Печально известный баг в команде деления с плавающей запятой (FDIV) в процессоре Pentium вынудил корпорацию Intel

отозвать чипы после того, как они были поставлены заказчикам, что стоило Intel 475 млн долларов. Моделирование необходимо для тестирования системы до того, как она будет выпущена.

Рисунок 4.1 показывает графики сигналов из модели предыдущего модуля `sillyfunction`, демонстрирующие, что модуль работает корректно[1]. В соответствии с логическим выражением y принимает значение логической 1, когда a, b и c принимают значения 000, 100 или 101.

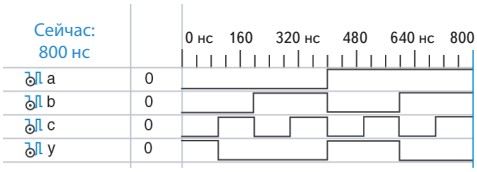
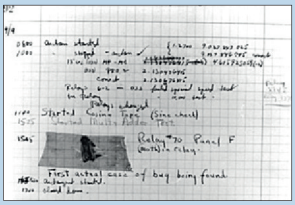


Рис. 4.1 Графики сигналов

Термин «баг» существовал еще до изобретения компьютера. В 1878 году Томас Эдисон называл багами «огрехи и затруднения» в своих изобретениях. Первый настоящий компьютерный баг был молью, попавшей между контактами реле электромеханического компьютера Harvard Mark II в 1947 году. Ее нашла Грейс Хоппер, которая зарегистрировала этот случай в рабочем журнале, приклеив моль и прокомментировав: «впервые обнаружен настоящий баг».



(Источник: запись в регистрационном журнале Исторического центра военно-морского флота, Флот США, фото № NII 96566-KN.)

Синтез

Логический синтез преобразует код на HDL в нетлист, описывающий цифровую аппаратуру (т. е. логические элементы и соединнения между ними). Логический синтезатор может выполнять оптимизацию для сокращения количества необходимых элементов. Нетлист может быть текстовым файлом или представлен в виде схемы, чтобы было легче визуализировать цифровую систему. **Рисунок 4.2** показывает результаты синтеза модуля `sillyfunction`[2]. Обратите внимание, что три трехвходовых элемента И упрощены в 2 двухвходовых элемента И, как было показано в **примере 2.6**, используя булеву алгебру.

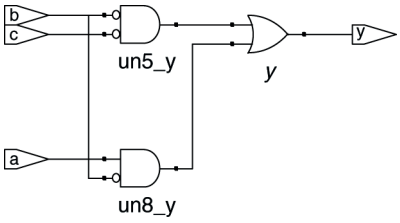


Рис. 4.2 Схема `sillyfunction`

Описание схем на HDL напоминает программный код. Но вы должны помнить, что этот код предназначен для описания аппаратуры.

[1] Моделирование было проведено в программе ModelSim PE Student Edition версии 10.0. Modelsim был выбран, так как он используется в коммерческих проектах и имеет студенческую версию с возможностью бесплатного моделирования до 10 тыс. строк кода. – *Прим. перев.*

[2] Синтез был сделан с помощью программы Synplify Premier от Synplicity. Этот САПР был выбран, так как он является лидирующим коммерческим продуктом для синтеза HDL в программируемые логические интегральные схемы (раздел 5.6.2) и так как он доступен по цене и подходит для использования в университетах. – *Прим. перев.*

SystemVerilog и VHDL – сложные языки со множеством операторов. Не все из операторов синтезируются в аппаратуре: например, оператор вывода результатов на экран во время моделирования не превращается в цифровую схему. Так как наша основная задача – создание цифровой схемы, акцент будет сделан на синтезируемом подмножестве языков. Точнее, мы будем делить код на HDL на синтезируемые модули и тестбенч. Синтезируемые модули описывают цифровую схему. Тестбенч содержит код, который подает воздействия на входы модуля и проверяет правильность значений его выходов, а также выводит несоответствия между ожидаемыми и действительными значениями. Код тестбенча предназначается только для моделирования и не может быть синтезирован.

Одна из главных ошибок начинающих заключается в том, что они думают о коде на HDL как о компьютерной программе, а не как об описании цифровой аппаратуры. Если вы не представляете, хотя бы примерно, во что должен синтезироваться ваш код на HDL, то, скорее всего, результат вам не понравится. Ваша цифровая схема может получиться гораздо больше, чем нужно, или может оказаться, что ваш код моделируется правильно, но не может быть реализован в аппаратуре. Вместо этого вы должны думать над вашей разработкой в терминах блоков комбинационной логики, регистров и конечных автоматов. Нарисуйте эти блоки на бумаге и покажите, как они будут подключены до того, как вы начнете разрабатывать код.

По нашему опыту, лучший способ выучить HDL – тренироваться на примерах. В HDL есть определенные способы описания разных типов логики; эти способы называются идиомами. В данной главе мы научим вас, как описывать идиомы для блоков каждого типа логики и затем как сложить блоки вместе, чтобы получить работающую систему. Когда вам понадобится описать аппаратуру определенного типа, посмотрите на похожий пример и адаптируйте его под свои цели. Мы не будем пытаться строго описывать весь синтаксис HDL, так как это скучно и ведет к представлению о HDL как о языках программирования, а не как о подспорье для разработки аппаратуры. Если вам понадобится дополнительная информация об особенностях языков, то обратитесь к спецификациям VHDL и SystemVerilog, изданным IEEE, или многочисленным сухим, но исчерпывающим учебникам (см. рекомендованный список литературы в конце книги).

4.2. Комбинационная логика

Помните, что мы тренируемся разрабатывать синхронные последовательные схемы, которые состоят из комбинационной логики и регистров. Состояние выходов комбинационной схемы зависит только от входных сигналов. В этом разделе описано, как создавать поведенческие модели комбинационной логики с использованием HDL.

4.2.1. Битовые операторы

Битовые операторы манипулируют однобитовыми сигналами или многоразрядными шинами. Так, модуль `inv` в **HDL-примере 4.2** описывает 4 инвертора, подключенных к четырехразрядным шинам.

HDL-пример 4.2 ИНВЕРТОРЫ

SystemVerilog

```
module inv(input logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` представляет собой 4-битную шину. Биты, от старшего к младшему, записываются так: `a[3]`, `a[2]`, `a[1]` и `a[0]`. Такой порядок битов называется *little-endian**, т. к. младший бит имеет наименьший битовый номер. Мы могли бы назвать шину `a[4:1]`, и тогда `a[4]` был бы старшим. Или мы могли бы написать `a[0:3]`, и тогда порядок битов от старшего к младшему был бы следующим: `a[0]`, `a[1]`, `a[2]` и `a[3]`. Такой порядок битов называется *big-endian*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

В VHDL для определения шин типа `STD_LOGIC` используется `STD_LOGIC_VECTOR`. `STD_LOGIC_VECTOR(3 downto 0)` представляет собой 4-битную шину. Биты от старшего к младшему: `a(3)`, `a(2)`, `a(1)` и `a(0)`. Такой порядок битов называется *little-endian*, т. к. младший бит имеет наименьший битовый номер. Мы могли бы объявить шину как `STD_LOGIC_VECTOR(4 downto 1)`, и тогда 4-й бит был бы старшим. Или мы могли бы написать `STD_LOGIC_VECTOR(0 to 3)`, тогда порядок битов от старшего к младшему был бы следующим: `a(0)`, `a(1)`, `a(2)` и `a(3)`. Такой порядок битов называется *big-endian*.

* Английский термин *little-endian* можно перевести как «оканчивающийся на младший». – *Прим. перев.*

Порядок следования разрядов шины является чисто условным (о происхождении термина рассказывается в **разделе 6.2.2**). Действительно, в этом примере порядок битов неважен, т. к. для набора инверторов не имеет значения, где какой бит находится. Порядок битов имеет значение только для некоторых операторов, например для оператора сложения, в котором сумма из одного столбца переносится в другой. Любой порядок является приемлемым, если он используется последовательно. Мы будем постоянно использовать порядок битов слева направо от старшего к младшему, `[N-1:0]` на языке SystemVerilog и `(N-1 downto 0)` на языке VHDL для N-разрядной шины.

После каждого примера кода в этой главе приводится схема, созданная из кода SystemVerilog средствами синтеза. **Рисунок 4.3** показывает, что модуль `inv` синтезируется в виде блока из 4 инверторов, обозначенных символом инвертора с надписью `y[3:0]`. Блок инверторов соединен с четырехбитными входной и выходной шинами. Подобная аппаратная реализация получается из синтезированного VHDL-кода.

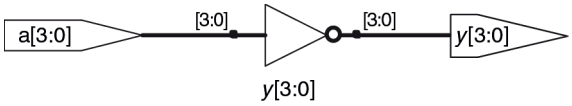


Рис. 4.3 Синтезированная схема модуля `inv`

Модуль `gates` в **HDL-примере 4.3** описывает битовые операции, которые выполняются на четырехбитных шинах для других основных логических функций.

HDL-пример 4.3 ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ

SystemVerilog

```
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3,
                    y4, y5);

  /*пять разных двухвходовых ЛЭ
    работают на 4-битных шинах */
  assign y1 = a & b;    // AND
  assign y2 = a |b;     // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a |b);  // NOR
endmodule
```

Символы ~, ^ и | – это примеры операторов в языке SystemVerilog, тогда как `a`, `b` и `y1` являются операндами. Комбинация операторов и операндов, такая как `a & b` или `~(a | b)`, называется *выражением*. Полная команда, такая как `assign y4 = ~(a & b);`, называется *оператором*.

`assign out = in1 op in2;` называется *оператором непрерывного присваивания*. Он заканчивается точкой с запятой. Когда в операторе непрерывного присваивания входные значения справа от знака = меняются, результат слева от знака = вычисляется заново. Таким образом, непрерывное присваивание описывает комбинационную логику.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- пять разных двухвходовых ЛЭ
  -- работают на 4-битных шинах
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

NOT, XOR и OR – это примеры операторов в языке VHDL, тогда как `a`, `b` и `y1` являются операндами. Комбинация операторов и операндов, такая как `a and b` или `a nor b`, называется *выражением*. Полная команда, например `y4 <= a nand b;`, называется *оператором*.

`out <= in1 op in2;` называется *оператором одновременного присваивания сигнала*. Операторы присваивания в VHDL заканчиваются точкой с запятой. Когда в операторе одновременного присваивания сигнала входные значения справа от знака <= изменяются, результат слева от знака <= вычисляется заново. Таким образом, оператор одновременного присваивания сигнала описывает комбинационную логику.

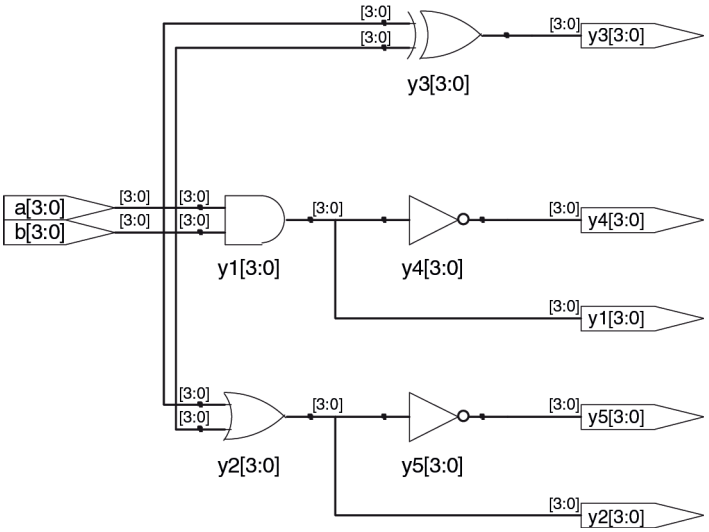


Рис. 4.4 **Синтезированная схема модуля `gates`**

4.2.2. Комментарии и пробелы

Пример с модулем `gates` демонстрирует, как оформлять комментарии. Языки SystemVerilog и VHDL не имеют особых требований к использованию свободного пространства (пробелов, табуляций и разрывов строк). Тем не менее надлежащие отступы и использование пустых строк помогают сделать читаемыми необычные конструкции. Будьте последовательны в использовании прописных букв и подчеркиваний в именах сигналов и модулей. В коде модуля `gates` используются только строчные буквы. Имена сигналов и модулей не должны начинаться с цифр.

SystemVerilog

Комментарии в языке SystemVerilog схожи с комментариями языков C или Java. Комментарии, начинающиеся с `/*`, могут занимать несколько строк, до следующего знака `*/`. Комментарии, начинающиеся с `//`, продолжаются до конца строки.

SystemVerilog чувствителен к регистру символов (прописным и строчным буквам). `y1` и `Y1` в SystemVerilog – это разные сигналы, но использование множества сигналов, отличающихся только регистром символов, вносит путаницу.

VHDL

Комментарии, начинающиеся с `/*`, могут занимать несколько строк до следующего знака `*/`. Комментарии, начинающиеся с `--`, продолжаются до конца строки.

VHDL не чувствителен к регистру символов. В VHDL `y1` и `Y1` – это один и тот же сигнал. Но другие программы, открывающие ваш файл, могут оказаться чувствительны к регистру символов, что приводит к неприятным ошибкам, если вы смешиваете прописные и строчные буквы.

4.2.3. Операторы сокращения

Операторы сокращения соответствуют многовходовым элементам, работающим на одной шине. **HDL-пример 4.4** описывает восьмивходовый логический элемент И с входами a_7, a_6, . . . , a_0. Аналогичные операторы сокращения существуют для логических элементов ИЛИ, Исключающее ИЛИ, И-НЕ, ИЛИ-НЕ и Исключающее ИЛИ с инверсией. Запомните, что многовходовый логический элемент Исключающее ИЛИ осуществляет функцию контроля четности, возвращая значение ИСТИНА, если нечетное количество входов имеют состояние ИСТИНА.

HDL-пример 4.4 ВОСЬМИВХОДОВЫЙ ЛОГИЧЕСКИЙ ЭЛЕМЕНТ И

SystemVerilog

```
module and8(input logic [7:0] a,
            output logic      y);

  assign y = &a;

  // &a записать гораздо проще, чем
  // assign y = a[7] & a[6] & a[5] &
  //            a[4] & a[3] & a[2] &
  //            a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a записать гораздо проще, чем
  -- y <= a(7) and a(6) and a(5) and a(4)
  --  and a(3) and a(2) and a(1) and a(0);
end;
```

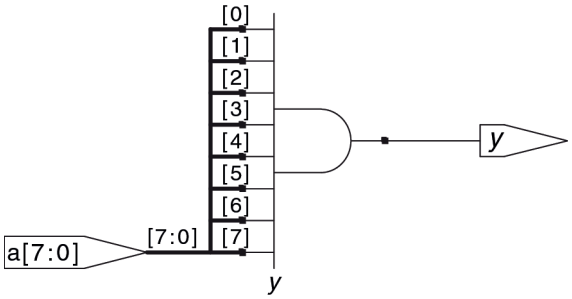


Рис. 4.5 Синтезированная схема модуля and8

4.2.4. Условное присваивание

Операторы условного присваивания выбирают один из нескольких указанных входов на основе состояния входа, называемого УСЛОВИЕ, и присваивают выбранный вход выходу. В **HDL-примере 4.5** показан двухвходовый мультиплексор, использующий условное присваивание.

HDL-пример 4.5 ДВУХВХОДОВЫЙ МУЛЬТИПЛЕКСОР

SystemVerilog

Условный оператор `?:` выбирает между вторым и третьим выражениями, руководствуясь первым выражением. Первое выражение называется *условие* (condition). Если условие принимает значение 1, то оператор выбирает второе выражение. Если условие принимает значение 0, то оператор выбирает третье выражение. Оператор `?:` особенно полезен для описания мультиплексоров, т. к. на основании состояния первого входа он выбирает между двумя другими. Следующий код демонстрирует программную реализацию двухвходового мультиплексора с 4-битными входами и выходами с использованием условного оператора.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

Если `s` равно 1, то `y = d1`, иначе `y = d0`. Оператор `?:` также называют *тернарным оператором*, так как он имеет три входа. С такой же целью он используется в языках C и Java.

VHDL

Условное присваивание сигнала осуществляет разные операции в зависимости от некоторого условия. Условные присваивания особенно полезны для описания мультиплексоров. Например, двухвходовый мультиплексор может использовать условное присваивание сигнала для выбора одного из двух 4-битных входов.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
       s:      in STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto
0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

Условное присваивание сигнала устанавливает `y` в `d1`, если `s` имеет значение 1. В противном случае оно устанавливает `y` в `d0`. Обратите внимание, что в версиях VHDL до 2008 нужно было писать `when s = '1'`, а не `when s`.

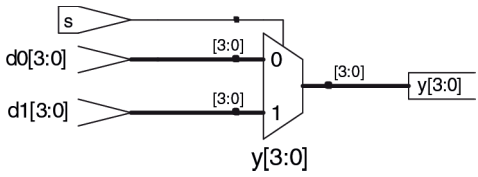


Рис. 4.6 Синтезированная схема модуля `mux2`

В **HDL-примере 4.6** описан четырехвходовый мультиплексор, работающий по тому же принципу, что и мультиплексор из **примера 4.5**.

На **рис. 4.7** изображена схема, созданная с помощью средств синтеза. Это программное обеспечение использует обозначение мультиплексора, отличающееся от того, которое до сих пор приводилось в тексте. Мультиплексор имеет многоразрядные входы данных (`d`) и одиночные входы разрешения (`e`). Когда один из входов активирован, соответствующие данные отправляются на выход. Например, когда `s[1] = s[0] = 0`,

нижний логический элемент И – `un1_s_5`, формирует 1, активируя нижний вход мультиплексора, в результате выбирается `d0[3:0]`.

HDL-пример 4.6 ЧЕТЫРЕХВХОДОВЫЙ МУЛЬТИПЛЕКСОР

SystemVerilog

Четырехвходовый мультиплексор может выбрать один из четырех входов с помощью вложенных условных операторов.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

Если `s[1]` принимает значение 1, тогда мультиплексор выбирает первое выражение, `(s[0] ? d3 : d2)`. Это выражение, в свою очередь, выбирает или `d3`, или `d2` на основе `s[0]` (`y = d3`, если `s[0]` имеет значение 1, и `d2`, если `s[0]` имеет значение 0). Если `s[1]` имеет значение 0, тогда мультиплексор подобным образом выбирает второе выражение, которое дает или `d1`, или `d0` в зависимости от `s[0]`.

VHDL

Четырехвходовый мультиплексор может выбрать один из четырех входов с помощью нескольких условий `else` в операторе условного присваивания сигнала.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;

end;
```

VHDL также поддерживает операторы выборочного присваивания сигнала для обеспечения более краткой записи, когда выбирается одна из нескольких возможностей. Это аналогично использованию операции `switch/case` вместо нескольких операций `if/else` в некоторых языках программирования. Четырехвходовый мультиплексор может быть переписан с использованием выборочного присваивания сигнала следующим образом:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

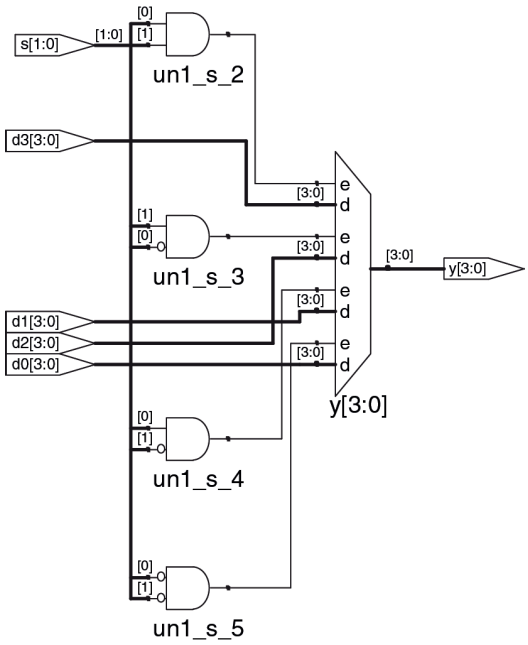


Рис. 4.7 **Синтезированная схема модуля `mux4`**

4.2.5. Внутренние переменные

Часто бывает удобно разделить сложную функцию на несколько промежуточных. Например, полный сумматор, который будет описан в **разделе 5.2.1**, представляет собой схему с тремя входами и двумя выходами, определяемыми следующими уравнениями:

$$\begin{aligned} S &= A \oplus B \oplus C_{\text{in}}; \\ C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}}. \end{aligned} \tag{4.1}$$

Если мы введем промежуточные сигналы P и G

$$\begin{aligned} P &= A \oplus B; \\ G &= AB, \end{aligned} \tag{4.2}$$

то сможем переписать уравнения для полного сумматора в виде:

$$\begin{aligned} S &= P \oplus C_{\text{in}}; \\ C_{\text{out}} &= G + PC_{\text{in}}. \end{aligned} \tag{4.3}$$

Переменные P и G называются *внутренними*, потому что они не являются ни входами, ни выходами, они используются только внутри модуля. Они подобны локаль-

Вы можете проверить это, заполнив таблицу истинности, чтобы убедиться, что это правильно.

ным переменным в языках программирования. **HDL-пример 4.7** показывает, как эти переменные используются в HDL.

Операции присваивания в HDL (`assign` в языке SystemVerilog и `<=` в VHDL) выполняются параллельно. Это отличается от традиционных языков программирования, таких как C или Java, в которых операторы выполняются в том порядке, в котором они записаны. В традиционных языках важно, чтобы выражение $S = P \oplus C_{in}$ следовало за выражением $P = A \oplus B$, поскольку операторы выполняются последовательно. В HDL порядок записи не имеет значения. Подобно аппаратным средствам, операторы присваивания HDL выполняются в момент, когда входы и сигналы с правой стороны выражения меняют свое значение независимо от порядка, в котором операторы присваивания появляются в модуле.

HDL-пример 4.7 ПОЛНЫЙ СУММАТОР

SystemVerilog

В языке SystemVerilog внутренние сигналы обычно объявляются как `logic`.

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g |(p & cin);
endmodule
```

VHDL

В VHDL для представления внутренних переменных обычно используются *сигналы*, значения которых определяются одновременными операторами присваивания, такими как `p <= a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;
  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

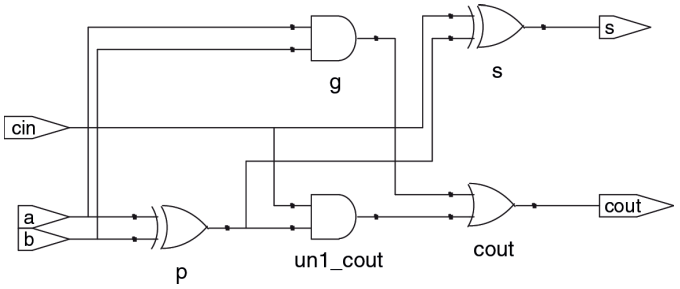


Рис. 4.8 Синтезированная схема модуля `fulladder`

4.2.6. Приоритет

Обратите внимание, что мы использовали скобки в вычислении C_{out} в **HDL-примере 4.7**, чтобы определить порядок операций: $C_{out} = G + (P \cdot C_{in})$, а не $C_{out} = (G + P) \cdot C_{in}$. Если мы не используем скобки, порядок операций определяется по умолчанию. **HDL-пример 4.8** определяет приоритет операторов от высшего к низшему для каждого языка. Таблицы включают арифметические операции, операции сдвига и операции сравнения, которые будут рассмотрены в **главе 5**.

HDL-пример 4.8 ПРИОРИТЕТ ОПЕРАТОРОВ

SystemVerilog

Таблица 4.1

	Операция	Значение
Высший	`~`	Побитовое отрицание (НЕ)
	`*, /, %`	Умножение, деление, остаток
	`+, -`	Сложение, вычитание
	`<<, >>`	Сдвиг влево/вправо
	`<<<, >>>`	Арифметический сдвиг влево/вправо
	`<, <=, >, >=`	Сравнение на больше-меньше
Низший	`==, !=`	Сравнение на равенство
	`&, ~&`	И, И-НЕ
	`^, ~^`	Исключающее ИЛИ, Исключающее ИЛИ-НЕ
	`|, ~|`	ИЛИ, ИЛИ-НЕ
	`?:`	Условный оператор

Система приоритета операторов для SystemVerilog подобна системам, принятым в других языках программирования. В частности, И имеет приоритет над ИЛИ. Можно пользоваться приоритетом операторов, чтобы исключить использование круглых скобок.

```
assign cout = g |p & cin;
```

VHDL

Таблица 4.2

	Операция	Значение
Высший	`not`	НЕ
	`*, /, mod, rem`	Умножение, деление, модуль, остаток
	`+, -`	Сложение, вычитание
	`rol, ror, srl, sll`	Циклический сдвиг влево/вправо Логический сдвиг влево/вправо
Низший	`<, <=, >, >=`	Сравнение на больше-меньше
	`=, /=`	Сравнение на равенство
	`and, or, nand, nor, xor, xnor`	Логические операции

В VHDL умножение имеет приоритет над сложением. Но, в отличие от SystemVerilog, здесь все логические операторы (`and`, `or` и т. д.) имеют одинаковый приоритет. Поэтому скобки необходимы; в противном случае `cout <= g or p and cin` будет интерпретироваться слева направо как `cout <= (g or p) and cin`.

4.2.7. Числа

Числа указываются в двоичной, восьмеричной, десятичной или шестнадцатеричной системе счисления (с основаниями 2, 8, 10 и 16 соответственно). Размер, т. е. количество битов, может быть также указан; свободные разряды заполняются нулями. Подчеркивания в числах игно-

рируются и могут быть полезными, когда требуется разделить длинное число на более читаемые фрагменты. **HDL-пример 4.9** объясняет, как числа записываются в каждом из языков.

HDL-пример 4.9 ЧИСЛА

SystemVerilog

Формат для объявления констант – `N'Bvalue`, где `N` – размер в битах, `B` – буква, указывающая на основание, и `value` – значение. Например, `9'h25` определяет 9-битное число со значением $25_{16} = 37_{10} = 000100101_2$. SystemVerilog поддерживает `'b` для основания 2, `'o` – для основания 8, `'d` – для основания 10 и `'h` – для основания 16. Если основание опущено, то по умолчанию оно равно 10.

Если не указан размер, то предполагается, что число содержит столько же битов, сколько и выражение, в котором оно используется. Недостающие старшие разряды дополняются нулями автоматически до полного размера. Например, если `w` – 6-битная шина, то `assign w = 'b11` присваивает `w` значение 000011. Лучшей практикой является явное указание размера. Исключением является то, что `'0` и `'1` служат конструкциями SystemVerilog для заполнения шины нулями или единицами соответственно.

VHDL

В VHDL числа STD_LOGIC записываются в бинарном коде и заключаются в одинарные кавычки: `'0'` и `'1'` указывают на логические уровни 0 и 1. Формат объявления констант типа `STD_LOGIC_VECTOR` следующий: `NB"value"`, где `N` – размер в битах, `B` – буква, указывающая на основание, и `value` – значение. Например, `9X"25"` определяет 9-битное число со значением $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 поддерживает `B` для основания 2, `O` – для основания 8, `D` – для основания 10 и `X` – для основания 16.

Если основание опущено, то по умолчанию оно равно 2. Если размер не указан, то предполагается, что число имеет размер, соответствующий количеству битов значения. По состоянию на октябрь 2011 SynplifyPremier от Synopsys не поддерживает указание размера.

`others = '0'` и `others = '1'` – конструкции VHDL с заполнением всех битов нулями или единицами соответственно.

Таблица 4.3

Запись	Кол-во битов	Основание	Значение	Представление
`3'b101`	3	2	5	101
`'b11`	?	2	3	0000011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	000101010

Таблица 4.4

Запись	Кол-во битов	Основание	Значение	Представление
`3B"101"`	3	2	5	101
`B"11"`	2	2	3	11
`8B"11"`	8	2	3	00000011
`8B"1010_1011"`	8	2	171	10101011
`3D"6"`	3	10	6	110
`6O"42"`	6	8	34	100010
`8X"AB"`	8	16	171	10101011
`"101"`	3	2	5	101
`B"101"`	3	2	5	101
`X"AB"`	8	16	171	10101011

4.2.8. Z-состояние и X-состояние

В HDL z-состояние используется для описания высокоимпедансного состояния. Использование z-состояния, в частности, полезно для описания буфера с тремя состояниями, состояние выхода которого является высокоимпедансным (отключенным), когда на вход разрешения подан 0. Вспомните из **раздела 2.6.2**, что шина может управляться несколькими буферами с тремя состояниями, только один из которых должен быть активен. **HDL-пример 4.10** демонстрирует программную реализацию тристабильного буфера. Если этот буфер активирован, то состояние на выходе будет таким же, как и на входе. Если буфер не активирован, то состояние на выходе является высокоимпедансным (`z`).

HDL-пример 4.10 ТРИСТАБИЛЬНЫЙ БУФЕР

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Обратите внимание, что `y` объявляется как `tri`, а не `logic`. Сигналы типа `logic` могут иметь только один драйвер. Тристабильные шины могут иметь несколько драйверов, поэтому они должны объявляться как `net`. Два применяемых типа `net` в SystemVerilog имеют названия `tri` и `trireg`. Обычно только один драйвер в сети активен в конкретный момент времени, и сеть принимает задаваемые им значения. Если ни один из драйверов не активирован, то `tri` находится в высокоимпедансном состоянии (`z`), в то время как `trireg` сохраняет предыдущее значение. Если для входа или выхода тип не указан, то предполагается, что тип – `tri`. Также обратите внимание, что выход модуля типа `tri` может использоваться как вход типа `logic` для других модулей. В дальнейшем цепи с несколькими драйверами будут рассматриваться в **разделе 4.7**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= a when en else "ZZZZ";
end;
```

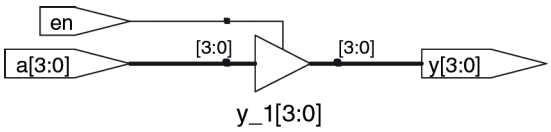


Рис. 4.9 Синтезированная схема модуля `tristate`

Также в HDL используют x для описания неопределенного логического уровня. Если на шину одновременно попадает 0 и 1 с двух активных тристабильных буферов (или других элементов), то в результате получаем x, что указывает на конфликт. Если все тристабильные буферы, управляющие шиной, одновременно находятся в состоянии OFF, то на шине будет высококимпедансное состояние, на что указывает z. В начале моделирования состояния узлов, таких как выходы триггеров, инициализируются неизвестным состоянием (x в SystemVerilog и u – в VHDL). Это помогает отслеживать ошибки, которые появляются, если вы забыли установить триггер в начальное состояние, перед тем как использовать его выход.

HDL-пример 4.11 ТАБЛИЦЫ ИСТИННОСТИ С НЕОПРЕДЕЛЕННЫМИ И ВЫСОКОИМПЕДАНСНЫМИ ВХОДАМИ

SystemVerilog

Сигналы в SystemVerilog могут принимать значения 0, 1, z и x. Константы SystemVerilog, начинающиеся с z или x, при необходимости дополняются символами z или x в старших разрядах (вместо нулей) для достижения необходимой длины.

Таблица 4.5 описывает таблицу истинности для логического элемента И, используя все четыре возможных значения сигнала. Обратите внимание, что логический элемент может иногда определять выход, несмотря на неизвестное состояние некоторых входов. Например, 0&z возвращает 0, потому что на выходе логического элемента И всегда 0, если какой-то из входов имеет состояние 0. В противном случае плавающее или некорректное состояние на входах приводит к неопределенным состояниям на выходах, обозначающимся в SystemVerilog как x.

Таблица 4.5

И		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

VHDL

Сигналы типа STD_LOGIC в VHDL могут принимать значения '0', '1', 'z', 'x' и 'u'.

Таблица 4.6 описывает таблицу истинности для логического элемента И, используя пять возможных значений сигнала. Обратите внимание, что логический элемент может иногда определять выход, несмотря на неизвестные состояния некоторых входов. Например, '0' и 'z' возвращает '0', т. к. на выходе логического элемента И всегда '0', если какой-то из входов имеет состояние '0'. В противном случае высокоимпедансное или неопределенное состояние на входах приводит к неопределенным состояниям на выходах, обозначаемых в VHDL как 'x'. Неинициализированные состояния входов приводят к неинициализированным состояниям сигналов на выходах, обозначаемым в VHDL как 'u'.

Таблица 4.6

И		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

Если логический элемент получает высокоимпедансное значение на входе, то он может сформировать x на выходе, когда у него не получается определить правильное выходное значение. Если элемент получает на входе неопределенное или неинициализированное значение, то на

выходе он может сформировать x. **HDL-пример 4.11** показывает, как в SystemVerilog и VHDL комбинируют эти различные значения сигналов в логических элементах. x- или u-состояния при моделировании практически всегда означают ошибки или плохой стиль программирования. В синтезированной цепи это соответствует плавающему входу элемента, неинициализированному состоянию или конфликту. x или u могут быть случайно интерпретированы схемой как 0 или 1, что приведет к непредсказуемому поведению программы.

4.2.9. Манипуляция с битами

Часто программистам приходится работать с фрагментом шины или сцеплять (объединять) сигналы для формирования шин. Эти операции называются манипуляциями с битами. В **HDL-примере 4.12** y задается 9-битной переменной $c_2c_1d_0d_0d_0c_0101$ с использованием манипуляций с битами.

HDL-пример 4.12 МАНИПУЛЯЦИИ С БИТАМИ

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Оператор {} используется для объединения шин. {3{d[0]}} указывает на три копии d[0]. Не путайте 3-битную двоичную константу 3'b101 с шиной с именем b. Обратите внимание, что определение длины 3-битной константы имеет решающее значение; в противном случае в середине y могло бы появиться неизвестное количество нулей. Если бы размерность y превышала 9 бит, то нули были бы помещены в старших битах.

VHDL

```
y <= (c(2 downto 1), d(0), d(0), d(0), c(0),
3B"101");
```

Оператор агрегирования используется для объединения шин. y должен быть 9-битным сигналом типа STD_LOGIC_VECTOR.

Другой пример демонстрирует возможности оператора агрегирования в VHDL. Предположим, что z – это 8-битный сигнал типа STD_LOGIC_VECTOR, тогда при выполнении операции агрегирования

```
z <= ("10", 4 => '1', 2 downto 1 =>'1',
others =>'0')
```

z получит значение 10010110. "10" переходит в старшую пару битов. 1 также помещается в 4-й бит и биты 2 и 1. Все остальные биты равны 0.

4.2.10. Задержки

Операторы в HDL могут быть связаны с задержками, указанными в произвольных единицах. В процессе моделирования задержки помогают предсказать, насколько быстро будет работать схема (если вы укажете адекватные задержки). Также при отладке они помогают понять причину и следствие (устанавливать источник плохого результата сложно, если в процессе моделирования все сигналы меняются одновременно). Эти задержки игнорируются в процессе синтеза; задержка элемента, сгенерированного синтезатором, зависит от значений t_{pd} и t_{cd}, а не от чисел в HDL-коде.

В **HDL-примере 4.13** добавлена задержка к первоначальной функции из **HDL-примера 4.1**, $y = \bar{a}\,\bar{b}\,\bar{c} + a\bar{b}\,\bar{c} + a\bar{b}c$. Предполагается, что инвертор имеет задержку 1 нс, трехвходовый элемент И имеет задержку 2 нс, а трехвходовый элемент ИЛИ – задержку 4 нс. **Рисунок 4.10** показывает результаты моделирования с задержкой сигнала `y` 7 нс относительно входов. Обратите внимание, что `y` неизвестно в начале моделирования.

HDL-пример 4.13 ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ С ЗАДЕРЖКАМИ

SystemVerilog

```
'timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 |n2 | n3;
endmodule
```

Файлы SystemVerilog могут включать директиву определения единицы модельного времени для указания, какому промежутку времени соответствует одна единица времени. Эта директива имеет вид `'timescale unit/precision`. В этом файле каждая единица времени равна 1 нс, а моделирование проводится с точностью 1 пс. Если в файле нет директивы установки модельного времени времени, то для единиц времени и точности используются значения по умолчанию (обычно оба параметра равны 1 нс). В SystemVerilog символ # используется для указания количества единиц задержки. Он может содержаться в операции `assign`, а также в неблокирующих (<=) и блокирующих (=) присваиваниях, которые будут рассмотрены в **разделе 4.5.4**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

В VHDL заявление `after` используется для обозначения задержек. Единицы в этом случае определяются в наносекундах.

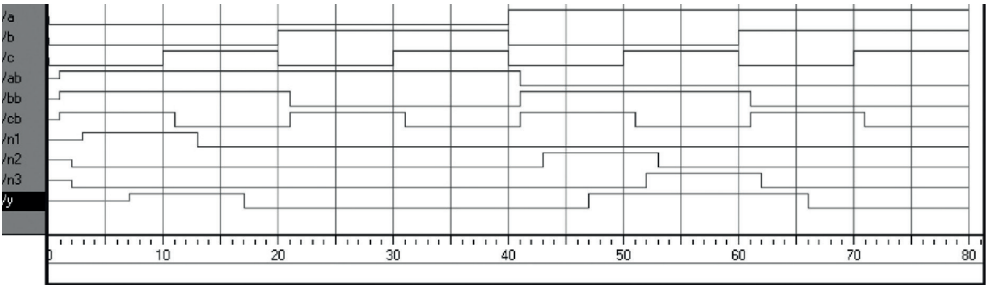


Рис. 4.10 Пример моделирования сигналов с задержками (среда моделирования ModelSim)

4.3. Структурное моделирование

В предыдущей главе обсуждалось *поведенческое* моделирование, описывающее модуль с точки зрения отношений между входами и выходами. Эта глава изучает *структурное* моделирование, описывающее модуль с точки зрения того, как он составлен из более простых модулей.

Например, **HDL-пример 4.14** показывает, как собирается четырехвходовый мультиплексор из трех двухвходовых мультиплексоров. Каждая копия двухвходового мультиплексора называется *экземпляром*. Множество экземпляров одного модуля различаются отдельными названиями. В данном примере это `lowmux`, `highmux` и `finalmux`. Это пример системы, в которой двухвходовый мультиплексор повторно используется много раз.

HDL-пример 4.14 СТРУКТУРНАЯ МОДЕЛЬ ЧЕТЫРЕХВХОДОВОГО МУЛЬТИПЛЕКСОРА

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

Три экземпляра модуля `mux2` называются `lowmux`, `highmux` и `finalmux`. Модуль `mux2` должен быть где-нибудь объявлен в SystemVerilog-коде (**HDL-примеры 4.5**, **4.15** или **4.34**).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

В архитектуре в первую очередь должны быть объявлены порты `mux2` при помощи оператора объявления компонента. Это позволяет инструментам VHDL проверить, что компонент, который вы хотите использовать, имеет те же порты, что и интерфейс, который был объявлен где-то еще в другом операторе интерфейса. Это дает возможность предотвратить ошибки, вызванные изменением интерфейса, но не самого объекта.

HDL-пример 4.14 (окончание)

При этом объявление компонента делает VHDL-код довольно громоздким.

Обратите внимание, что эта архитектура модуля `mux4` была названа `struct`, тогда как архитектуры модулей с поведенческими описаниями из **раздела 4.2** назывались `synth`. VHDL позволяет иметь множество архитектур (реализаций) одного интерфейса; архитектуры различаются по имени. Сами имена не имеют значения для инструментов САПР, но `struct` и `synth` являются общепринятыми. Синтезируемый VHDL-код, как правило, содержит только одну архитектуру для каждого интерфейса, так что мы не будем обсуждать VHDL-синтаксис, используемый для настройки того, какую архитектуру выбирать, когда определено множество из них.

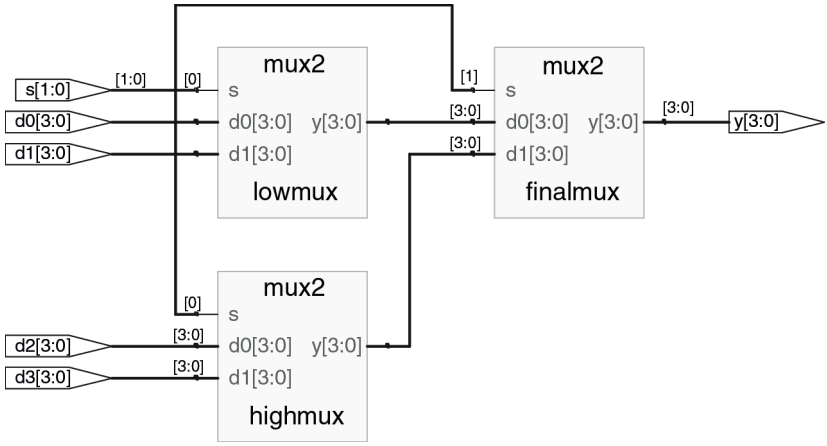


Рис. 4.11 Синтезированная схема модуля `mux4`

В **HDL-примере 4.15** используется структурное моделирование для создания двухвходового мультиплексора из пары буферов с тремя состояниями. Но строить логические схемы из таких буферов не рекомендуется.

HDL-пример 4.15 СТРУКТУРНАЯ МОДЕЛЬ ДВУХВХОДОВОГО МУЛЬТИПЛЕКСОРА

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

В языке SystemVerilog, такие как `~s` выражения разрешены в списке портов экземпляра. Допустимы выражения любой сложности, но это не поощряется, потому что они делают код сложным для чтения.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out  STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

В языке VHDL такие выражения, как `not s`, не разрешены в карте портов экземпляра. Таким образом, `sbar` должен быть определен как отдельный сигнал.

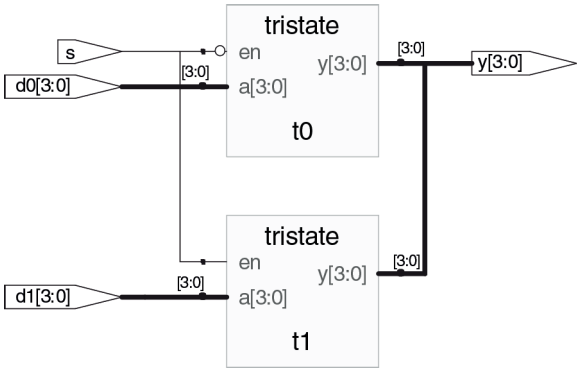


Рис. 4.12 **Синтезированная схема модуля `mux2`**

В **HDL-примере 4.16** показано, как модули могут получать доступ к части шины. Двухвходовый мультиплексор разрядностью 8 бит построен с помощью двух четырехбитных двухвходовых мультиплексоров, объявленных ранее и работающих с младшим и старшим полубайтами.

HDL-пример 4.16 ОБРАЩЕНИЕ К ЧАСТЯМ ШИН

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto
0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto
0));
  end component;
begin
  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

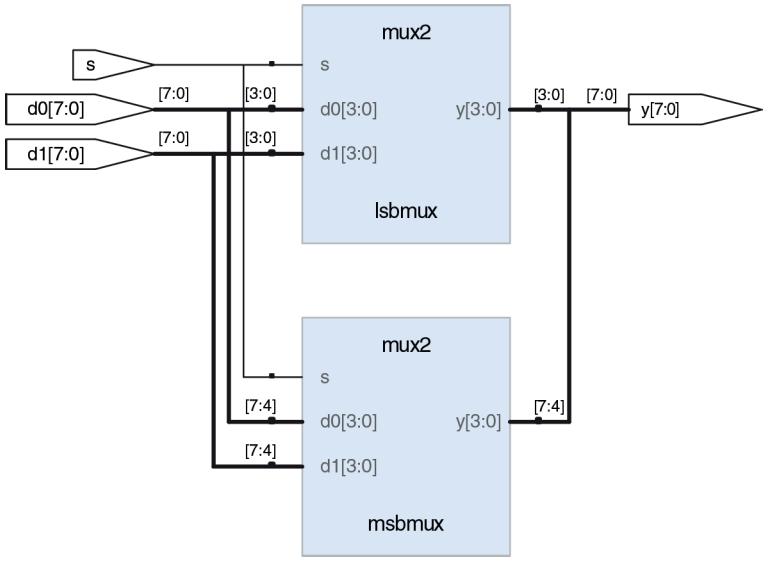


Рис. 4.13. Синтезированная схема модуля `mux2_8`

Обычно все сложные системы создаются иерархически. Система описывается структурно с помощью включения в нее основных компонентов.

Каждый из этих компонентов описывается структурно из своих строительных блоков и так далее рекурсивно до тех пор, пока дело не дойдет до частей, достаточно простых для поведенческого описания. Хорошим стилем является стремление избежать (или по крайней мере минимизировать) смешения структурных и поведенческих описаний внутри одного модуля.

4.4. Последовательностная логика

Синтезаторы HDL распознают определенные идиомы и превращают их в конкретные последовательностные схемы. Код, разработанный в ином стиле, может быть правильно смоделирован, но в синтезированной схеме могут оказаться как грубые, так и труднораспознаваемые ошибки. В этом разделе представлены идиомы, рекомендованные для описания регистров и защелок.

4.4.1. Регистры

Подавляющее большинство современных коммерческих систем построено на регистрах, использующих срабатывающие по переднему фронту тактового импульса D-триггеры. В **HDL-примере 4.17** показана идиома для такого триггера.

Сигналы, значения которым присвоены в операторах `always` языка SystemVerilog и операторах `process` языка VHDL, сохраняют свое состояние, пока не случится событие из списка чувствительности оператора, приводящее к изменению их состояния. Поэтому код, использующий эти операторы с соответствующими списками чувствительности, может описывать последовательностные схемы с памятью. Например, у триггера в списке чувствительности есть только сигнал `clk`, и потому триггер хранит старое значение `q` до следующего переднего фронта `clk`, даже если входной сигнал `d` изменился раньше.

В отличие от операторов `always` и `process`, оператор непрерывного присваивания SystemVerilog (`assign`) и оператор одновременного присваивания VHDL (<=) перевычисляются каждый раз, когда изменяется какая-либо из переменных в правой части, поэтому эти операторы могут описать только комбинационную логику[1].

4.4.2. Регистры со сбросом

В начале моделирования или сразу после подачи питания на схему значения на выходе триггеров или регистров неизвестны, что обозначается как значение `x` в SystemVerilog или как `u` в VHDL. На практике полезно использовать регистры с входом сброса, чтобы при включении можно было привести систему в начальное определенное состояние. Сброс может быть синхронным или асинхронным. Помните, что асинхронный сброс

[1] С помощью этих операторов можно описывать и логику, сохраняющую состояние, например `assign q = clk ? d : q;` но делать это не рекомендуется. – *Прим. перев.*

происходит немедленно, в отличие от синхронного, который устанавливает в 0 выходной сигнал только по следующему переднему фронту такта. В **HDL-примере 4.18** показаны идиомы для триггеров с асинхронным и синхронным сбросами. Следует учитывать, что отличить синхронный и асинхронный сбросы на принципиальной схеме может быть непросто. Например, некоторые средства синтеза помещают на схемах асинхронный сброс на нижней стороне триггера, а синхронный – на левой.

HDL-пример 4.17 РЕГИСТР

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

В общем случае оператор `always` языка SystemVerilog имеет вид

```
always @(sensitivity list)
  statement;
```

Оператор выполняется, только когда случается событие, заданное в списке чувствительности. В этом примере оператором является `q <= d` (читается «q принимает значение d»). Таким образом, триггер копирует `d` в `q` по переднему фронту тактового сигнала, а в остальное время значение `q` остается неизменным. Список чувствительности также иногда называют списком стимулов.

`<=` называется *неблокирующим присваиванием*. Пока считайте его обычным присваиванием `=`; мы вернемся к трудноуловимой разнице между ними в **разделе 4.5.4**. Заметьте, что внутри оператора `always` неблокирующее присваивание `<=` используется вместо `assign`.

Как мы увидим в последующих разделах, операторы `always` можно использовать для создания триггеров, защелок или комбинационной логики в зависимости от списка чувствительности и оператора. Из-за подобной гибкости языка при синтезе аппаратных блоков можно непреднамеренно получить нежелательную конфигурацию. Для избежание таких ошибок в SystemVerilog введены операторы `always_ff`, `always_latch` и `always_comb`. Оператор `always_ff` ведет себя так же, как `always`, но используется только тогда, когда подразумевается синтез триггеров, и позволяет инструментальной среде в противном случае выдавать предупреждение.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

Оператор `process` языка VHDL имеет вид:

```
process(sensitivity list) begin
  statement;
end process;
```

Оператор выполняется, когда изменяется какая-либо из переменных из списка чувствительности. В этом примере оператор `if` проверяет, было ли изменение передним фронтом тактового сигнала (такта) `clk`. Если да, то `q <= d` (читается «q принимает значение d»). Таким образом, триггер копирует `d` в `q` по переднему фронту сигнала `clk`, а в остальное время значение `q` остается неизменным.

Другой вариант идиомы VHDL для записи триггера:

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

`rising_edge(clk)` является синонимом `clk'event and clk = '1'`.

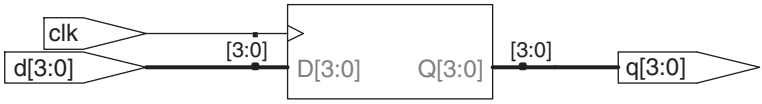


Рис. 4.14 **Синтезированная схема модуля** `flop`

HDL-пример 4.18 РЕГИСТР СО СБРОСОМ

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // асинхронный сброс
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // синхронный сброс
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

Сигналы в списке чувствительности оператора `always` разделяются запятой или словом `or`. Заметьте, что у триггера с асинхронным сбросом в списке чувствительности есть сигнал `posedge reset`, а у триггера с синхронным сбросом этого сигнала нет. Поэтому триггер с асинхронным сбросом реагирует на передний фронт `reset` немедленно, а с синхронным – только по переднему фронту такта.

В примере у обоих модулей одно и то же имя `flopr`, поэтому в схеме можно использовать либо один модуль, либо другой.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Сигналы в списке чувствительности оператора `process` разделяются запятой. Заметьте, что у триггера с асинхронным сбросом в списке чувствительности `reset` есть, а у триггера с синхронным сбросом – нет. Поэтому триггер с асинхронным сбросом реагирует на передний фронт `reset` немедленно,

HDL-пример 4.18 (окончание)

а с синхронным – только по переднему фронту такта.

Помните, что состояние триггера инициализируется как `'u'` при старте моделирования VHDL. Как уже упоминалось, имя архитектуры (в данном примере `synchronous` или `asynchronous`) игнорируется инструментальной средой, но помогает людям, читающим код.

Так как обе архитектуры описывают один и тот же объект `flopr`, в схеме можно использовать либо одну архитектуру, либо другую.

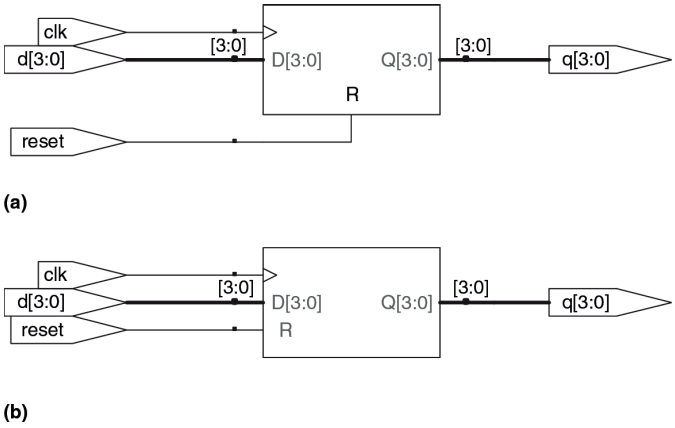


Рис. 4.15 Синтезированная схема модуля `flopr`: (а) с асинхронным сбросом, (б) с синхронным сбросом

4.4.3. Регистры с сигналом разрешения

Регистры с сигналом разрешения реагируют на тактовый импульс только при условии подачи логической единицы на линию разрешения.

В **HDL-примере 4.19** показан регистр с разрешающим входом `en` и асинхронным сбросом `reset`, сохраняющий предыдущее значение, если оба сигнала имеют значение FALSE.

HDL-пример 4.19 РЕГИСТР С УСЛОВИЕМ И СБРОСОМ

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // асинхронный сброс
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- асинхронный сброс
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

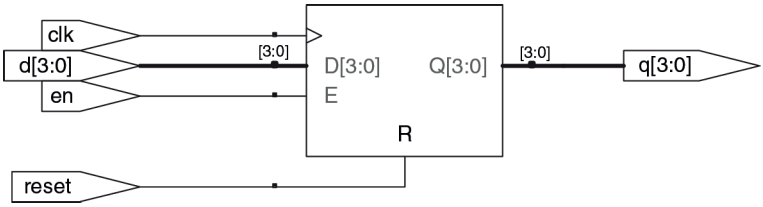


Рис. 4.16 **Синтезированная схема модуля** `flopenr`

4.4.4. Группы регистров

Один оператор `always/process` можно использовать для описания нескольких элементов аппаратуры. Рассмотрим, например, синхронизатор из **раздела 3.5.5**, состоящий из двух последовательных триггеров, показанный на **рис. 4.17** и описанный в **HDL-примере 4.20**. По переднему фронту `clk`, `d` копируется в `n1`, и в то же время `n1` копируется в `q`.

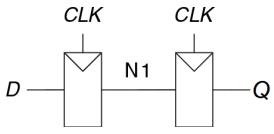


Рис. 4.17 **Схема синхронизатора**

HDL-пример 4.20 СИНХРОНИЗАТОР

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d; // неблокирующее
               // присваивание
      q <= n1; // неблокирующее
    end
endmodule
```

Обратите внимание на конструкцию `begin/end`. Она является операторными скобками для группы из нескольких операторов, находящихся внутри оператора `always`, наподобие скобок {} в C или Java. Конструкция `begin/end` не была нужна в примере `flopr`, потому что `if/else` считается одним оператором.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

Переменная `n1` должна быть объявлена как `signal`, так как она используется внутри модуля в качестве сигнала для соединения логических элементов.

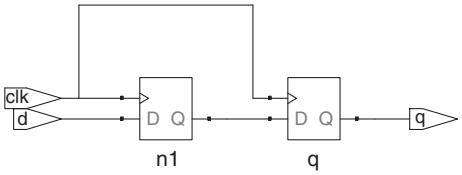


Рис. 4.18 Синтезированная схема модуля `sync`

4.4.5. Защелки

Возвращаясь к **разделу 3.2.2**, вспомним, что D-защелка открыта при высоком уровне тактового сигнала, т. е. пропускает сигнал данных с входа на выход. Защелка закрывается, когда уровень становится низким, сохраняя свое значение. Фрагмент кода в **HDL-примере 4.21** показывает идиому для D-защелки.

Не все программы-синтезаторы хорошо справляются с защелками. Если вы не уверены, что ваш синтезатор их поддерживает, или нет особых причин использовать именно защелки, пользуйтесь вместо них триггерами, работающими по фронту сигнала. Также нужно следить, чтобы в коде на HDL не было конструкций, приводящих к появлению нежелательных защелок, что легко может произойти в результате невнима-

тельности. Многие программы синтеза предупреждают, когда создают защелку; и если вы ее не ждали, то ищите ошибку в своем коде. А если вы не знаете, нужна ли вам в схеме защелка или нет, то это, скорее всего, значит, что вы ведете разработку на HDL как на обычном языке программирования и у вас впереди могут быть большие проблемы.

HDL-пример 4.21 D-ЗАЩЕЛКА

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` в данном случае эквивалентно `always @(clk, d)` и оптимально для описания защелки на SystemVerilog. Оператор `always_latch` вычисляется при каждом изменении `clk` или `d`.

При высоком уровне `clk` переменная `q` принимает значение `d`, т. е. этот код описывает защелку, активную по высокому уровню.

В противном случае `q` сохраняет свое значение. SystemVerilog может выдавать предупреждение, если оператор `always_latch` не описывает реальную защелку.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

В списке чувствительности есть и `clk`, и `d`, так что `process` вычисляется каждый раз, когда `clk` или `d` изменяется. При высоком уровне `clk` переменная `q` принимает значение `d`.

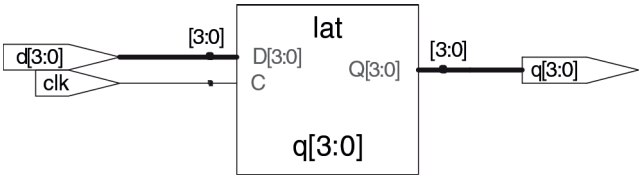


Рис. 4.19. Синтезированная схема модуля `latch`

4.5. И снова комбинационная логика

В **разделе 4.2** мы использовали операторы присваивания для поведенческого описания комбинационной логики. Операторы `always` языка SystemVerilog и операторы `process` языка VHDL используются для описания последовательностных схем, потому что они сохраняют состояние переменных, если не было указано их изменить. Но эти операторы можно использовать и для поведенческого описания комбинационной логики, если список чувствительности описан так, чтобы отвечать на любое изменение входных сигналов, и тело оператора определяет значение выходного

сигнала при любой комбинации значений входов. Код на HDL в **HDL-примере 4.22** использует операторы `always/process` для описания группы из четырех инверторов (синтезированная схема приведена на **рис. 4.3**).

HDL-пример 4.22 ИНВЕРТОР, РЕАЛИЗОВАННЫЙ С ПОМОЩЬЮ `always/process`

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic[3:0] y);
  always_comb
    y = ~a;
endmodule
```

Оператор `always_comb` выполняет выражения внутри оператора `always` каждый раз, когда изменяется любой из сигналов в правой части <= или = оператора `always`. В данном случае это эквивалентно `always@(a)`, но гораздо надежнее, так как позволяет избегать ошибок в случае переименования или добавления сигналов в оператор `always`.

Если код внутри оператора `always_comb` не является комбинационной логикой, то тогда SystemVerilog будет выдавать предупреждение. Оператор `always_comb` эквивалентен `always@(*)`, но является более предпочтительным в SystemVerilog. Равенство = в операторе `always` называется *блокирующим присваиванием*, в отличие от неблокирующего присваивания <=. В SystemVerilog хорошей практикой является использование блокирующих присваиваний для комбинационной логики и неблокирующих – для последовательностной. Это будет далее обсуждаться в **разделе 4.5.4**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

Оператор `process(all)` исполняет все выражения внутри `process`, как только изменяется любой из сигналов оператора `process`. Это эквивалентно `process(a)`, но существенно лучше, так как позволяет избежать ошибок при переименовании или добавлении новых сигналов.

Операторы `begin` и `end process` обязательны в VHDL, даже если `process` содержит только одно присваивание.

В обоих языках можно использовать блокирующие и неблокирующие присваивания в операторах `always/process`. Внутри одного оператора блокирующие присваивания выполняются в том порядке, в котором они написаны, в точности как в обычном языке программирования, а обновление значений переменных в левой части неблокирующих присваиваний выполняется «одновременно», после того как вычислены значения всех правых частей неблокирующих присваиваний.

Код в **HDL-примере 4.23** описывает полный сумматор, в котором использованы промежуточные сигналы `p` и `g` для вычисления `s` и `cout`. В результате получается та же схема, что и на **рис. 4.8**, но с использованием операторов `always/process` вместо операторов присваивания.

Эти два примера не очень удачны для демонстрации использования `always/process` для комбинационной логики – в них больше строк кода, чем в эквивалентных **HDL-примерах 4.2** и **4.7** с использованием операторов присваивания. Но для моделирования более сложной комбинационной логики удобно пользоваться операторами `case` и `if`, которые допускаются только внутри операторов `always/process`. Их мы рассмотрим в следующих разделах.

SystemVerilog

В операторе `always` знак равенства = означает блокирующее присваивание, а <= означает неблокирующее (также известное как одновременное) присваивание.

Не путайте эти два присваивания с непрерывным присваиванием с помощью оператора `assign`. Операторы `assign` должны использоваться вне операторов `always` и тоже вычисляются одновременно.

VHDL

В операторе `process` := означает блокирующее присваивание, а <= означает неблокирующее (одновременное) присваивание.

Неблокирующие присваивания применяются к выходам и к сигналам. Блокирующие присваивания применяются к переменным, объявленным в операторах `process` (код в **HDL-примере 4.23**). Символ <= может использоваться и за пределами операторов `process`, где тоже исполняется одновременно.

HDL-пример 4.23 ПОЛНЫЙ СУММАТОР, РЕАЛИЗОВАННЫЙ С ИСПОЛЬЗОВАНИЕМ ОПЕРАТОРОВ `always/process`

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p = a ^ b;           // блокирующее
      g = a & b;           // блокирующее
      s = p ^ cin;         // блокирующее
      cout = g |(p & cin); // блокирующее
    end
endmodule
```

Здесь эквивалентом `always_comb` было бы `always @(a, b, cin)`, но `always_comb` лучше, поскольку позволяет избежать ошибок, связанных с недостающими в списке чувствительности сигналами.

По причинам, которые мы обсудим в **разделе 4.5.4**, для комбинационной логики лучше использовать блокирующие присваивания. В этом примере они использованы для вычисления вначале `p`, затем `g`, `s` и `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- блокирующее
    g := a and b; -- блокирующее
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Здесь эквивалентом оператора `process(all)` был бы `process(a, b, cin)`, но `process(all)` лучше, поскольку позволяет избежать ошибок, связанных с недостающими в списке чувствительности сигналами.

HDL-пример 4.23 (окончание)

Так как `p` и `g` упоминаются в левой части операторов блокирующего присваивания (`:=`) в операторе `process`, то они должны быть объявлены как `variable`, а не как `signal`. Объявление переменных пишется перед `begin` того процесса, в котором эти переменные используются.

4.5.1. Операторы case

Рассмотрим еще один пример использования операторов `always/process` для комбинационной логики – дешифратор для семисегментного индикатора, выполненный с использованием оператора `case`, который должен появляться внутри оператора `always/process`.

Согласно **примера 2.10** дешифратора семисегментного индикатора процесс разработки больших блоков комбинационной логики утомителен и чреват ошибками. Языки описания аппаратуры HDL облегчают этот процесс, позволяя определять функциональность на более высоком уровне абстракции и затем автоматически синтезировать ее в логические элементы. В коде **HDL-примера 4.24** используется оператор `case` для описания дешифратора семисегментного индикатора по таблице истинности. Оператор `case` выполняет различные действия в зависимости от значения своих входных данных. Он подразумевает комбинационную логику, если все возможные сочетания входных данных определены; в противном случае получится последовательностная логика, и выход сохранит свое предыдущее значение в неопределенных случаях.

Средства синтеза синтезируют дешифратор семисегментного индикатора как постоянную память (ПЗУ), содержащую 7 выходов для каждой из 16 возможных комбинаций входов. ПЗУ обсуждаются в **разделе 5.5.6**.

Если бы условие `default` или `others` не было упомянуто в операторе `case`, то дешифратор сохранял бы предыдущее значение выхода, когда вход находится в диапазоне 10–15. Для аппаратуры такое поведение было бы странно.

Обычные дешифраторы часто записываются с помощью операторов `case`. В коде **HDL-примера 4.25** представлен дешифратор 3:8.

HDL-пример 4.24 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
     //                    abc_defg
     0:       segments = 7'b111_1110;
     1:       segments = 7'b011_0000;
     2:       segments = 7'b110_1101;
     3:       segments = 7'b111_1001;
     4:       segments = 7'b011_0011;
     5:       segments = 7'b101_1011;
     6:       segments = 7'b101_1111;
     7:       segments = 7'b111_0000;
     8:       segments = 7'b111_1111;
     9:       segments = 7'b111_0011;
     default: segments = 7'b000_0000;
   endcase
endmodule
```

Оператор `case` проверяет значение `data`. Если `data` равно 0, выполнится действие после двоеточия, т. е. `segments` установится в 1111110. Аналогично проверяются другие значения `data` вплоть до 9 (обратите внимание, что по умолчанию система счисления десятичная). Условие `default` – удобный способ определить выход для всех случаев, не перечисленных явно, гарантируя комбинационную логику. В SystemVerilog операторы `case` обязаны находиться внутри операторов `always`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                          abcdefg
      when X"0" =>   segments <= "1111110";
      when X"1" =>   segments <= "0110000";
      when X"2" =>   segments <= "1101101";
      when X"3" =>   segments <= "1111001";
      when X"4" =>   segments <= "0110011";
      when X"5" =>   segments <= "1011011";
      when X"6" =>   segments <= "1011111";
      when X"7" =>   segments <= "1110000";
      when X"8" =>   segments <= "1111111";
      when X"9" =>   segments <= "1110011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

В отличие от SystemVerilog, VHDL поддерживает операторы условного присваивания сигнала (**пример 4.6**), которые по сути похожи на операторы `case`, но могут встречаться и за пределами операторов `process`, так что поводов использовать операторы `process` для описания комбинационной логики в VHDL меньше.

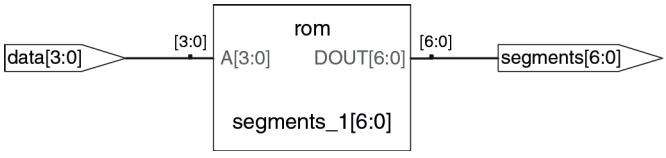


Рис. 4.20 Синтезированная схема модуля `sevenseg`

HDL-пример 4.25 ДЕШИФРАТОР 3:8

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);

  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

Строго говоря, условие `default` в данном случае для синтеза не нужно, поскольку перечислены все возможные сочетания входов, но оно полезно для моделирования на случай, если какой-либо из входов равен `x` или `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000" =>  y <= "00000001";
      when "001" =>  y <= "00000010";
      when "010" =>  y <= "00000100";
      when "011" =>  y <= "00001000";
      when "100" =>  y <= "00010000";
      when "101" =>  y <= "00100000";
      when "110" =>  y <= "01000000";
      when "111" =>  y <= "10000000";
      when others => y <= "XXXXXXXX";
    end case;
  end process;
end;
```

Строго говоря, условие `others` в данном случае не нужно для синтеза, поскольку перечислены все возможные сочетания входов, но оно полезно для моделирования на случай, если какой-либо из входов равен `x`, `z` или `u`.

4.5.2. Условный оператор (if)

Операторы `always/process` могут содержать также операторы `if`, за которыми может следовать оператор `else`. Если все возможные сочетания входов учтены условиями, то оператор описывает комбинационную логику, иначе – последовательностную (например, защелка в **разделе 4.4.5**).

В **HDL-примере 4.26** используются операторы `if` для описания схемы приоритетов, определенной в **разделе 2.4**. Вспомним, что *N*-входовая схема приоритетов устанавливает в значение TRUE тот из выходов, который соответствует наиболее приоритетному входу, равному TRUE.

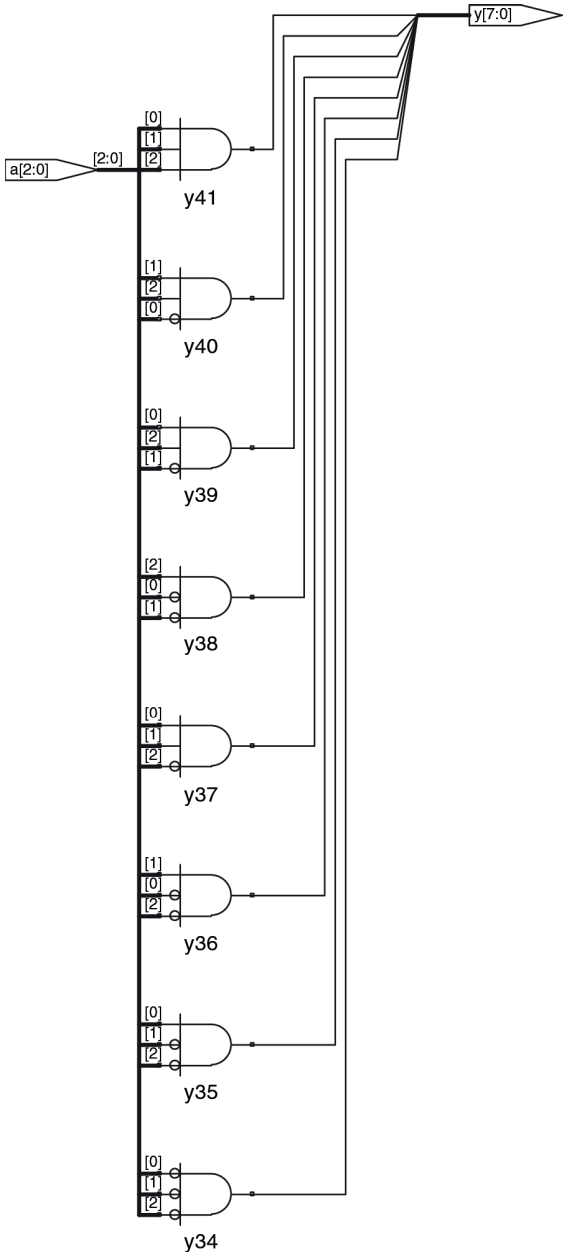


Рис. 4.21 **Синтезированная схема модуля** `decoder3_8`

HDL-пример 4.26 СХЕМА ПРИОРИТЕТОВ

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

В SystemVerilog операторы `if` обязаны быть внутри операторов `always`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if   a(3) then y <= "1000";
    elsifa(2) then y <= "0100";
    elsifa(1) then y <= "0010";
    elsifa(0) then y <= "0001";
    else           y <= "0000";
    end if;
  end process;

end;
```

В отличие от SystemVerilog, в VHDL есть операторы условного присваивания (**HDL-пример 4.6**), которые по сути похожи на операторы `if`, но могут встречаться и за пределами операторов `process`, так что поводов использовать процессы для описания комбинационной логики в VHDL меньше.

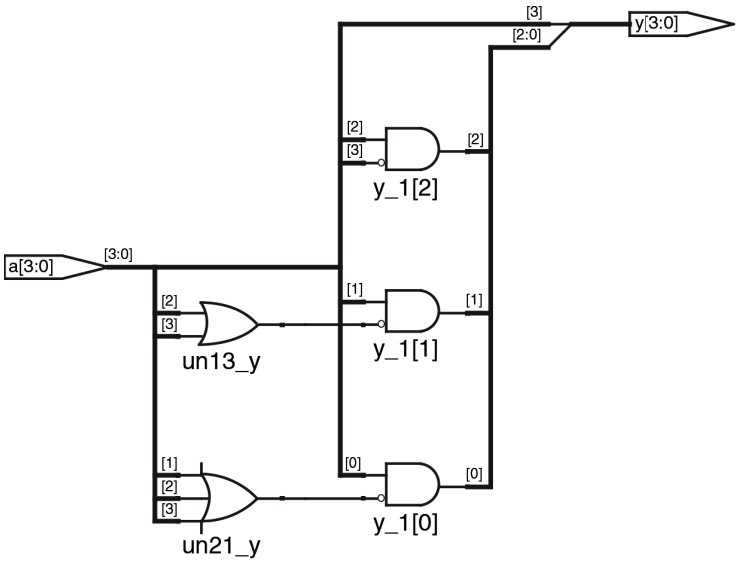


Рис. 4.22 Синтезированная схема модуля `prioritycкt`

4.5.3. Таблицы истинности с незначащими битами

Как показано в **разделе 2.7.3**, в таблицах истинности могут быть незначащие биты ради упрощения логики. В коде **HDL-примера 4.27** показано, как описать приоритетную схему с незначащими битами.

Средства синтеза для этого модуля генерируют схему, приведенную на **рис. 4.23**, которая немного отличается от схемы приоритетов на **рис. 4.22**, но они логически эквивалентны.

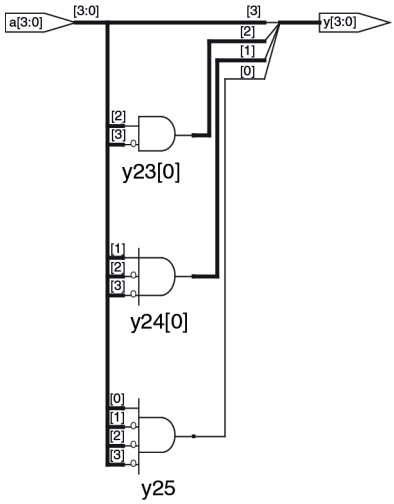


Рис. 4.23 Синтезированная схема модуля `priority_casez`

HDL-пример 4.27 СХЕМА ПРИОРИТЕТОВ С НЕЗНАЧАЩИМИ БИТАМИ

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

Оператор `casez` работает так же, как и `case`, но еще и распознает знак «?» как незначащий бит.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end

architecture dontcare of priority_casez is
begin
  process(all) begin
    casez a is
      when "1---" => y <= "1000";
      when "01--" => y <= "0100";
      when "001-" => y <= "0010";
      when "0001" => y <= "0001";
      when others => y <= "0000";
    end casez;
  end process;
end;
```

Оператор `casez` работает так же, как и `case`, но еще и распознает знак «-» как незначащий бит.

4.5.4. Блокирующие и неблокирующие присваивания

В кратком руководстве ниже объясняется, когда и как использовать тот или иной тип присваивания. Если ему не следовать, то можно разработать код, который, возможно, будет работать в режиме моделирования, но будет синтезироваться в некорректную схему. Далее в этом разделе объясняются принципы, лежащие в основе данного руководства.

РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ БЛОКИРУЮЩИХ И НЕБЛОКИРУЮЩИХ ПРИСВАИВАНИЙ

SystemVerilog

1. Используйте `always_ff @(posedge clk)` и неблокирующие присваивания для моделирования последовательностной логики.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // неблокирующее
    q <= n1; // неблокирующее
  end
```

2. Используйте непрерывные присваивания для моделирования простой комбинационной логики.

```
assign y = s ? d1 : d0;
```

3. Используйте `always_comb` и блокирующие присваивания для моделирования более сложной комбинационной логики, когда удобнее использовать оператор `always`.

```
always_comb
  begin
    p = a ^ b; // блокирующее
    g = a & b; // блокирующее
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Не присваивайте значение одному и тому же сигналу в разных операторах `always` или непрерывных присваиваниях.

VHDL

1. Используйте `process(clk)` и неблокирующие присваивания для моделирования синхронной последовательностной логики.

```
process(clk) begin
  if rising_edge(clk) then
    n1 <= d; -- неблокирующее
    q <= n1; -- неблокирующее
  end if;
end process;
```

2. Используйте одновременные присваивания вне операторов `process` для моделирования простой комбинационной логики.

```
y <= d0 when s = '0' else d1;
```

3. Используйте `process(all)` для моделирования более сложной комбинационной логики, если оператор `process` удобнее. Пользуйтесь блокирующими присваиваниями для локальных переменных.

```
process(all)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- блокирующее
  g := a and b; -- блокирующее
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Не присваивайте значение одной и той же переменной в разных операторах `process` или одновременных присваиваниях.

Комбинационная логика

Полный сумматор в коде **HDL-примера 4.23** корректно смоделирован с использованием блокирующих присваиваний. В этом разделе мы рассмотрим, как он работает и чем он отличается от модели, использующей неблокирующие присваивания.

Представьте, что значения `a`, `b` и `cin` первоначально равны 0. Значения `p`, `g`, `s` и `cout` будут тоже равны 0. В какой-то момент `a` изменяется на 1, активируя оператор `always/process`. Четыре блокирующих присваивания выполняются в показанном ниже порядке. (В случае VHDL присваивания `s` и `cout` выполняются одновременно.) Заметьте, что `p` и `g` получают свои новые значения до вычисления `s` и `cout` из-за блокирующих присваиваний. Это важно, потому что мы хотим вычислять `s` и `cout`, пользуясь новыми значениями `p` и `g`.

1. `p` ← $1 \oplus 0 = 1$.
2. `g` ← $1 \cdot 0 = 0$.
3. `s` ← $1 \oplus 0 = 1$.
4. `cout` ← $0 + 1 \cdot 0 = 0$.

HDL-пример 4.28 демонстрирует использование неблокирующих присваиваний.

HDL-пример 4.28 ПОЛНЫЙ СУММАТОР С НЕБЛОКИРУЮЩИМИ ПРИСВАИВАНИЯМИ

SystemVerilog

```
// неблокирующие присваивания
// (не рекомендуется)

module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p <= a ^ b; // неблокирующее
      g <= a & b; // неблокирующее
      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule;
```

VHDL

```
-- неблокирующие присваивания
-- (не рекомендуется)

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process(all) begin
    p <= a xor b; -- неблокирующее
    g <= a and b; -- неблокирующее
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Так как `p` и `g` появляются в левой части неблокирующих присваиваний в операторе `process`, они должны быть объявлены как `signal`, а не как `variable`. Объявление `signal` появляется перед `begin` в `architecture`, а не в `process`.

Рассмотрим тот же случай, когда a из 0 становится 1, в то время как b и cin равны 0. Четыре неблокирующих присваивания выполняются одновременно:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad s \leftarrow 0 \oplus 0 = 0 \quad cout \leftarrow 0 + 0 \cdot 0 = 0.$$

Таким образом, s вычисляется одновременно с p, и потому использует старое значение p. Из-за этого s остается равным 0, а не становится 1. Но p при этом изменяется с 0 на 1. Это изменение вызывает исполнение оператора always/process во второй раз:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad s \leftarrow 1 \oplus 0 = 1 \quad cout \leftarrow 0 + 1 \cdot 0 = 0.$$

На этот раз p уже равно 1, и s, как и следует, становится равным 1. Неблокирующие присваивания в конце концов приходят к правильному ответу, но оператору always/process приходится выполняться дважды. От этого модель получается медленнее, хотя код и синтезируется в ту же схему.

Еще один недостаток неблокирующих присваиваний для моделирования комбинационной логики – при симуляции HDL может дать неверный результат, если забыть упомянуть промежуточные переменные в списке чувствительности[1].

Хуже того, некоторые синтезаторы создадут правильную схему, даже если неверный список чувствительности приводит к неверной модели. Это ведет к несовпадению результатов моделирования и реального поведения аппаратуры.

SystemVerilog

Если бы список чувствительности оператора always в коде **HDL-примера 4.28** был написан как always@(a, b, cin), а не как always_comb, оператор не выполнился бы повторно, когда изменились p или g. В этом случае s ошибочно остался бы равным 0 вместо 1.

VHDL

Если бы список чувствительности оператора process в коде **HDL-примера 4.28** был записан как process(a, b, cin), а не как process(all), оператор не выполнился бы повторно, когда изменились p или g. В этом случае s ошибочно остался бы равным 0 вместо 1.

Последовательностная логика

Синхронизатор в коде **HDL-примера 4.20** корректно смоделирован с использованием неблокирующих присваиваний. По переднему фронту тактового сигнала d копируется в n1 в то же время, как n1 копируется в q, так что код, как и следует, описывает два регистра. Например, пусть первоначально d = 0, n1 = 1 и q = 0. По переднему фронту тактового сигнала одновременно выполняются два присваивания, так что после прохождения фронта n1 = 0 и q = 1:

$$n1 \leftarrow d = 0 \quad q \leftarrow n1 = 1.$$

[1] При использовании always_comb и process(all) для комбинационной логики этот недостаток неактуален. – *Прим. перев.*

В коде **HDL-примера 4.29** делается попытка описать тот же модуль с помощью блокирующих присваиваний. По переднему фронту `clk`, `d` копируется в `n1`. Затем это новое значение `n1` копируется в `q`, в результате чего значение `d` ошибочно оказывается и в `n1`, и в `q`. Присваивания выполняются одно за другим, так что после фронта сигнала `q` = `n1` = 0.

1. `n1` ← `d` = 0.
2. `q` ← `n1` = 0.

Оттого, что переменная `n1` не видна окружающему миру и не влияет на поведение `q`, синтезатор ликвидирует ее в процессе оптимизации, как показано на **рис. 4.24**.

HDL-пример 4.29 ПЛОХОЙ СИНХРОНИЗАТОР С БЛОКИРУЮЩИМИ ПРИСВАИВАНИЯМИ

SystemVerilog

```
// Плохая реализация синхронизатора из-за
// применения блокирующих присваиваний

module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 = d; // блокирующее
      q = n1; // блокирующее
    end
endmodule
```

VHDL

```
-- Плохая реализация синхронизатора из-за
-- применения блокирующих присваиваний

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      n1 := d; -- блокирующее
      q <= n1;
    end if;
  end process;
end;
```

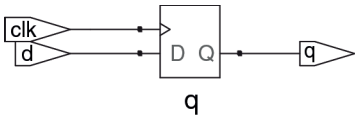


Рис. 4.24 Синтезированная схема для `syncbad`

Мораль этой иллюстрации такова: для моделирования последовательной логики в операторах `always/process` следует пользоваться исключительно неблокирующими присваиваниями. С помощью определенных хитростей, например изменения порядка присваиваний, можно добиться правильной работы блокирующих присваиваний, но они не дают никаких преимуществ, а лишь привносят риск нежелательного поведения. Неко-

торые последовательностные схемы не будут работать с использованием блокирующих присваиваний независимо от их порядка[1].

4.6. Конечные автоматы

Конечный автомат (КА) состоит из регистра состояния и двух блоков комбинационной логики для вычисления следующего состояния и выхода по заданному текущему состоянию и информации на входе, как показано на **рис. 3.22**. Описания конечных автоматов на HDL, соответственно, состоят из трех частей, моделирующих регистр состояния, логику следующего состояния и логику выхода.

В **HDL-примере 4.30** описывается КА деления на 3 из **раздела 3.4.2**. Для инициализации КА используется асинхронный сброс. Регистр состояния использует стандартную идиому для триггеров. Логика формирования следующего состояния и выхода является комбинационной.

HDL-пример 4.30 КОНЕЧНЫЙ АВТОМАТ, ДЕЛЯЩИЙ НА 3

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2}
statetype;
  statetype state, nextstate;

  // регистр состояния
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // логика следующего состояния
  always_comb
    case (state)
      S0:      nextstate = S1;
      S1:      nextstate = S2;
      S2:      nextstate = S0;
      default: nextstate = S0;
    endcase

  // выходная логика
  assign y = (state == S0);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- регистр состояния
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- логика следующего состояния
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- выходная логика
  y <= '1' when state = S0 else '0';
end;
```

[1] Авторы предлагают принять на веру, что не стоит использовать в SystemVerilog блокирующее присваивание для последовательностной логики, даже если оно в операторе always единственное. Это связано с особенностями алгоритмов моделирования SystemVerilog, в подробности которых мы не будем вдаваться. – *Прим. перев.*

HDL-пример 4.30 (окончание)

Оператор `typedef` определяет значение `statetype` как двухбитный `logic` тип с тремя возможными значениями: `S0`, `S1` или `S2`. `state` и `nextstate` – сигналы типа `statetype`.

Константам перечисления, упомянутым в определении типа, по умолчанию присваиваются порядковые значения: `S0` = `00`, `S1` = `01`, и `S2` = `10`. Они могут быть явно изменены пользователем, но программа-синтезатор рассматривает их как рекомендацию, а не как требование. Например, следующий фрагмент кодирует состояния трехбитным позиционным (one-hot) кодом:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 =
3'b010, S2 = 3'b100}
statetype;
```

Из-за того, что логика для следующего состояния должна быть комбинационной, условие `default` (значения по умолчанию) является обязательным даже несмотря на то, что состояния `2'b11` не бывает. Выход `y` равен 1, когда автомат находится в состоянии `S0`. Результат операции сравнения на равенство `a == b` равен 1, когда `a` равно `b`, и 0 в противном случае. Операция сравнения на неравенство `a != b`, наоборот, дает 1, когда `a` не равно `b`.

В этом примере определяется новый тип перечисляемых данных `statetype` с тремя возможными значениями: `S0`, `S1` и `S2`. `state` и `nextstate` – сигналы типа `statetype`. Благодаря использованию перечисления, а не явно задаваемых кодов состояний, VHDL позволяет синтезатору выбрать оптимальный код для состояний.

Выход `y` равен 1, когда `state` равно `S0`. Операция сравнения на неравенство записывается как `/=`. Чтобы получить на выходе 1, когда состояние отлично от `S0`, замените сравнение на `state /= S0`.

Средства синтеза генерируют лишь блочную диаграмму и диаграмму переходов для автомата; они не показывают логические элементы или входы и выходы на узлах и дугах, поэтому следует проверить по диаграмме, правильно ли вы определили КА в HDL-коде.

Диаграмма переходов на **рис. 4.25** для КА деления на 3 аналогична диаграмме на **рис. 3.28 (b)**. Двойной кружок означает, что при поступлении сигнала сброса автомат оказывается в состоянии S0. Реализация автомата на уровне логических элементов была показана в **разделе 3.4.2**.

Заметьте, что состояния обозначены константами перечисления, а не двоичными значениями. Благодаря этому код становится более читабельным и его легче изменять.

Если по какой-либо причине мы захотим, чтобы выход был равен 1 в состояниях S0 и S1, выходная логика изменится следующим образом:

Заметьте, что при синтезе использовано 3-битное кодирование (`Q[2:0]`), а не 2-битное кодирование, привычное для кода SystemVerilog.

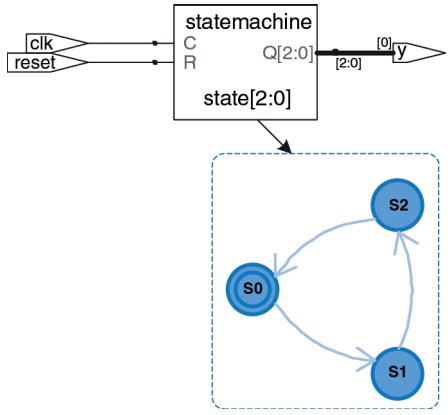


Рис. 4.25 Синтезированная схема модуля `divideby3FSM`

SystemVerilog

```
// выходная логика
assign y = (state== S0 | state== S1);
```

VHDL

```
-- выходная логика
y <= '1' when (state = S0 or state = S1) else '0';
```

Следующие два примера описывают КА распознавателя битового шаблона улитки из **раздела 3.4.3**. В коде показано, как использовать операторы `case` и `if` для обработки следующего состояния и выходной логики, зависящей и от входа, и от текущего состояния. В автомате Мура (**HDL-пример 4.31**) выход зависит только от текущего состояния, а в автомате Мили (**HDL-пример 4.32**) выход зависит и от текущего состояния, и от входов.

HDL-пример 4.31 АВТОМАТ МУРА ДЛЯ РАСПОЗНАВАНИЯ БИТОВОГО ШАБЛОНА

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2}
statetype;
  statetype state, nextstate;

  // регистр состояния
  always_ff @(posedge clk,
              posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // логика следующего состояния
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // выходная логика
  assign y = (state ==S2);
endmodule
```

Заметьте, что неблокирующие присваивания (<=) используются в регистре состояния для описания последовательностной логики, а для комбинационной логики следующего состояния используются блокирующие присваивания (=).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- регистр состояния
  process(clk, reset) begin
    if reset then              state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- логика следующего состояния
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- выходная логика
  y <= '1' when state = S2 else '0';
end;
```

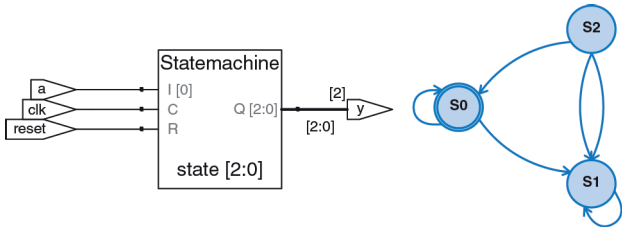


Рис. 4.26 Синтезированная схема модуля `patternMoore`

HDL-пример 4.32 АВТОМАТ МИЛИ ДЛЯ РАСПОЗНАВАНИЯ БИТОВОГО ШАБЛОНА

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // регистр состояния
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // логика следующего состояния
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default: nextstate = S0;
    endcase

  // выходная логика
  assign y = (a & state == S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- регистр состояния
  process(clk, reset) begin
    if reset then      state <= S0;
    elsif rising_edge(clk) then
                       state <= nextstate;
    end if;
  end process;

  -- логика следующего состояния
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- выходная логика
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

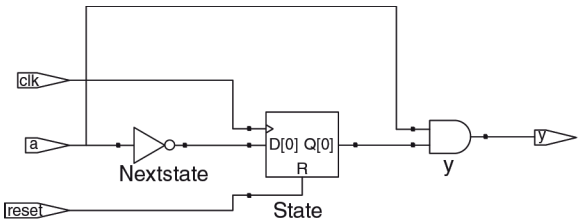


Рис. 4.27 Синтезированная схема модуля `patternMealy`

4.7. Типы данных

В этом разделе более подробно рассматриваются особенности типов данных в SystemVerilog и VHDL.

4.7.1. SystemVerilog

В предшественнике SystemVerilog, языке Verilog, в основном использовались два типа: `reg` и `wire`. Несмотря на свое название, сигнал типа `reg` не обязан соответствовать регистру, и эта путаница затрудняла изучение языка. Во избежание этой путаницы в SystemVerilog добавлен тип `logic`, который и используется в данной книге. В этом разделе подробно рассказывается о типах `reg` и `wire` для тех, кому предстоит читать старый код на языке Verilog.

В Verilog, если сигнал встречается в левой части оператора `<=` или `=` в `always`-блоке, он должен быть объявлен как `reg`, в противном случае – как `wire`. Поэтому сигнал типа `reg` может быть выходом триггера, защелки или комбинационной логики, в зависимости от списка чувствительности и оператора внутри `always`-блока.

У входных и выходных портов по умолчанию тип `wire`, если их тип не объявлен как `reg`. Ниже показано, как триггер описывается на обычном Verilog. Обратите внимание, что сигналы `clk` и `d` – типа `wire` по умолчанию, а `q` явно объявлен как `reg`, потому что он встречается в левой части оператора `<=` в `always`-блоке.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

Тип `logic`, добавленный в SystemVerilog, – это синоним типа `reg`, но его название избавлено от нежелательных ассоциаций с триггером. Кроме того, в SystemVerilog ослаблены ограничения в части использования операторов `assign` и в иерархических назначениях портов, так

что сигналы типа `logic` могут быть использованы вне блоков `always` – там, где традиционно требовались бы сигналы типа `wire`. Таким образом, подавляющее большинство сигналов в SystemVerilog может быть типа `logic`. Исключение – сигнал с несколькими источниками, например тристабильная высокоимпедансная шина с тремя состояниями. Такой сигнал должен быть объявлен как цепь (net), как показано в коде **HDL-примера 4.10**. Благодаря этому правилу, когда сигнал типа `logic` по ошибке подключен к нескольким источникам, SystemVerilog выдает сообщение об ошибке уже во время компиляции, а не присваивает ему значение `x` во время моделирования.

Наиболее распространенные типы цепей – `wire` и `tri`. Эти два типа – синонимы, но `wire` традиционно используется, когда источник один, а `tri` – когда их несколько. В SystemVerilog в типе `wire` нет необходимости: для сигналов с одним источником `logic` предпочтительнее.

Когда у всех активных источников цепи типа `tri` одно и то же значение, она получает это значение. Если все источники неактивны, цепь отключена (`z`). Если у активных источников разные значения (`0`, `1`, `x`), то цепь находится в состоянии конфликта (`x`).

Есть и другие типы цепей, значения которых определяются по-иному при неактивных источниках или в случае конфликта. Эти типы используются редко, но могут встречаться там же, где и тип `tri` (например, для цепей с несколькими источниками). Они описаны в **табл. 4.7**.

Таблица 4.7 Определение значения цепей

Тип цепи	Значение при неактивных источниках	Значение при конфликте источников
`tri`	`z`	`x`
`trireg`	`предыдущее значение`	`x`
`triand`	`z`	`0, если есть хоть один 0`
`trior`	`z`	`1, если есть хоть одна 1`
`tri0`	`0`	`x`
`tri1`	`1`	`x`

4.7.2. VHDL

В отличие от SystemVerilog, язык VHDL – со строгой типизацией, что защищает пользователя от некоторых ошибок, но временами он неуклюж.

Несмотря на то что тип `STD_LOGIC` принципиально важен, он не встроен в язык VHDL, а является частью библиотеки `IEEE.STD_LOGIC_1164`. Из-за этого в каждом файле должны быть операторы подключения библиотеки, что можно было видеть выше в примерах.

Кроме того, в IEEE.STD_LOGIC_1164 отсутствуют базовые операции типа сложения, сравнения, сдвигов и преобразования в целые из данных типа STD_LOGIC_VECTOR. Их, в конце концов, добавили в стандарте VHDL 2008 в библиотеку IEEE.NUMERIC_STD_UNSIGNED.

В VHDL также есть тип BOOLEAN с двумя значениями: true и false. Значения типа BOOLEAN возвращаются операциями сравнения (например, сравнения на равенство, s = '0') и используются в условных операторах, как when и if. Казалось бы, BOOLEAN true должно быть эквивалентно STD_LOGIC '1', а BOOLEAN false должно значить то же, что и STD_LOGIC '0', но эти типы не были взаимозаменяемы вплоть до VHDL 2008. Например, в старом коде на VHDL приходилось писать

```
y <= d1 when (s = '1') else d0;
```

а в VHDL 2008, где оператор when автоматически преобразует s из STD_LOGIC в BOOLEAN, уже можно писать просто

```
y <= d1 when s else d0;
```

Но и в VHDL 2008 все еще нужно писать

```
q <= '1' when (state = S2) else '0';
```

а не

```
q <= (state = S2);
```

потому что (state = S2) возвращает результат типа BOOELAN, который не может быть присвоен сигналу типа STD_LOGIC.

Хотя мы не объявляем никаких сигналов типа BOOLEAN, они автоматически являются результатом сравнения из сравнений и используются в условных операторах. Аналогично в VHDL есть тип INTEGER для представления целых чисел со знаком. Сигналы типа INTEGER могут принимать значения от $-(2^{31}-1)$ до $2^{31}-1$. В качестве индексов массивов нужно использовать целые числа, которые имеют тип INTEGER. Например, в операторе

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2 и 3 – целые типа INTEGER, служащие индексами для выбора битов сигнала a. Для индексации нельзя использовать сигнал типа STD_LOGIC или STD_LOGIC_VECTOR, поэтому нужно преобразовать его в INTEGER, как показано ниже в примере восьмивходового мультиплексора, выбирающего один бит из вектора с помощью трехбитного индекса. Функция TO_INTEGER, определенная в библиотеке IEEE.NUMERIC_STD_UNSIGNED, преобразует из STD_LOGIC_VECTOR в неотрицательные значения INTEGER.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mux8 is
  port(d: inSTD_LOGIC_VECTOR(7 downto 0);
```

```
       s: inSTD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;
architecture synth of mux8 is
begin
  y <= d(TO_INTEGER(s));
end;
```

VHDL также строг в отношении портов типа `out`: их можно использовать исключительно в качестве выходов. Например, следующий пример двух- и трехвходового логического элемента И некорректен, так как `v` – выход, но используется также для вычисления `w`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v, w: out STD_LOGIC);
end;
architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

Для решения этой проблемы в VHDL есть отдельный тип порта: `buffer`. Сигнал, подключенный к такому порту, ведет себя как выход, но также может быть использован внутри модуля. Вот исправленный текст объявления интерфейса:

```
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v: buffer   STD_LOGIC;
       w: out      STD_LOGIC);
 end;
```

В Verilog и SystemVerilog этого ограничения никогда не было, поэтому в них не нужны буферные порты. В VHDL 2008 это ограничение было также снято за счет разрешения доступа к чтению выходных портов.

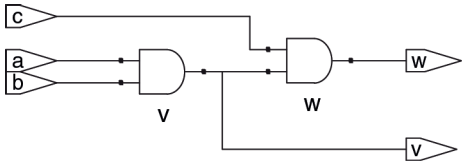


Рис. 4.28 Синтезированная схема модуля `and23`

В результате многих операций, таких как сложение, вычитание или операции булевой алгебры, получается одно и то же битовое представление результата, будь он со знаком или без знака. В отличие от них, сравнения на больше-меньше, умножение и арифметические сдвиги вправо

выполняются для чисел в дополнительном коде со знаком и двоичных чисел без знака по-разному. Эти операции рассматриваются в **главе 5**. В коде **HDL-примера 4.33** показано, как обозначаются сигналы, представляющие числа со знаком.

HDL-пример 4.33 БЕЗЗНАКОВЫЙ УМНОЖИТЕЛЬ (a) И УМНОЖИТЕЛЬ СО ЗНАКОМ (b)

SystemVerilog

```
// 4.33(a): беззнаковый умножитель
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);

  assign y = a *b;
endmodule

// 4.33(b): умножитель со знаком
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a *b;
endmodule
```

В SystemVerilog сигналы понимаются как беззнаковые по умолчанию. Добавление модификатора `signed` (например, `logic signed [3:0] a`) приводит к тому, что сигнал рассматривается как число со знаком.

VHDL

```
-- 4.33(a): беззнаковый умножитель
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

В VHDL для выполнения арифметических операций и операций сравнения над `STD_LOGIC_VECTOR` используется библиотека `NUMERIC_STD_UNSIGNED`. При этом векторы считаются беззнаковыми.

```
use IEEE.NUMERIC_STD_UNSIGNED.all
```

В VHDL также определены типы данных `UNSIGNED` и `SIGNED` (в библиотеке `IEEE.NUMERIC_STD`), но их рассмотрение выходит за рамки этой главы.

4.8. Параметризированные модули

До сих пор у модулей в наших примерах входы и выходы были фиксированной ширины. Например, нам понадобилось определить два разных модуля для двухразрядного мультиплексора с четырехразрядными и восьмиразрядными входами. Но в языках описания аппаратуры HDL можно описывать и параметризированные модули с портами переменной ширины.

В коде **HDL-примера 4.34** объявляется параметризированный двухразрядный мультиплексор с шириной входов, равной по умолчанию восьми битам, который затем используется для создания четырехразрядных мультиплексоров с восьмиразрядными и двенадцатиразрядными входами.

HDL-пример 4.34 ПАРАМЕТРИЗИРОВАННЫЕ N-БИТНЫЕ ДВУХРАЗРЯДНЫЕ МУЛЬТИПЛЕКСОРЫ

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input logic [width-1:0] d0, d1,
    input logic             s,
    output logic [width-1:0]y);

   assign y = s ? d1 : d0;
endmodule
```

В SystemVerilog возможна конструкция `#(parameter ...)` перед списком входов и выходов для определения параметров модуля. В примере выше оператор `parameter` состоит из параметра по имени `width` со значением по умолчанию, равным 8. Число разрядов на входах и выходах может зависеть от параметра.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

8-битный четырехразрядный мультиплексор состоит из трех экземпляров двухразрядного мультиплексора с шириной входов, установленной по умолчанию.

В отличие от него, в 12-битном четырехразрядном мультиплексоре `mux4_12` понадобится переопределить ширину входов с помощью конструкции `#( )` перед именем экземпляра (instance):

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Не путайте использование знака `#` для обозначения задержек с использованием `#(...)` при объявлении и переопределении параметров.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in STD_LOGIC_VECTOR(width-1 downto 0);
    s: in STD_LOGIC;
    y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

Оператор `generic` состоит из указания значения 8 по умолчанию для `width` типа `INTEGER`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(1 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux: mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

8-битный четырехразрядный мультиплексор, `mux4_8`, включает три мультиплексора 2:1 с шириной по умолчанию.

В отличие от него, в 12-битном четырехразрядном мультиплексоре `mux4_12` понадобится переопределить ширину по умолчанию с помощью `generic map`:

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

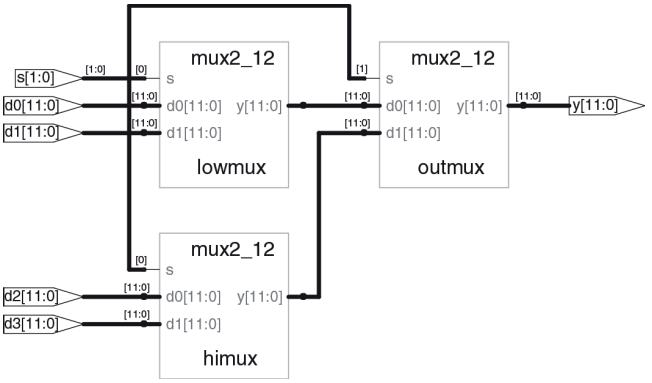


Рис. 4.29 Синтезированная схема модуля `mux4_12`

В коде **HDL-примера 4.35** показан дешифратор, который является еще более удачным примером параметризированного модуля. Широкий дешифратор $N{:}2^N$ довольно утомительно описывать с помощью оператора `case`, но это легко сделать с помощью параметризированного модуля, который просто устанавливает нужный бит в 1. Иначе говоря, в дешифраторе использовано блокирующее присваивание для установки всех битов в 0, а затем нужный бит изменяется в 1.

HDL-пример 4.35 ПАРАМЕТРИЗИРОВАННЫЙ ДЕШИФРАТОР $N{:}2^N$

SystemVerilog

```
module decoder

  #(parameter N = 3)
  (input logic [N-1:0]    a,
   output logic [2**N-1:0] y);
  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

`2**N` означает 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in STD_LOGIC_VECTOR(N-1 downto 0);
       y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

`2**N` означает 2^N.

В языках описания аппаратуры также предусмотрен оператор `generate` для получения разного количества аппаратуры в зависимости от значения параметра. В операторе `generate` допускаются циклы `for` и операторы `if` для определения количества и свойств желаемой аппа-

ратуры. В коде **HDL-примера 4.36** демонстрируется, как использовать операторы `generate` для получения *N*-входовой функции И из каскада двухвходовых логических элементов И. Конечно, для этой конкретной цели лучше подошла бы операция сокращения, но этот пример иллюстрирует общий принцип использования оператора `generate`.

Используйте операторы `generate` с осторожностью – из-за них можно легко непреднамеренно получить очень большую схему!

HDL-пример 4.36 ПАРАМЕТРИЗИРОВАННЫЙ *N*-ВХОДОВЫЙ ЛОГИЧЕСКИЙ ЭЛЕМЕНТ И

SystemVerilog

```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);
  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0] = a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i] = a[i] & x[i-1];
    end
  endgenerate

  assign y = x[width-1];
endmodule
```

Оператор `for` проходит по `i = 1, 2, ..., width-1` для получения множества последовательных логических элементов И. После `begin` в цикле `for` внутри `generate` должно быть двоеточие и произвольная метка (в данном случае `forloop`)*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
   signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
   x(0) <= a(0);
   gen: for i in 1 to width-1 generate
     x(i) <= a(i) and x(i-1);
   end generate;
   y <= x(width-1);
end;
```

Переменную цикла `generate` объявлять не нужно.

* Обратите также внимание на объявление переменной цикла `i` как `genvar`. – *Прим. перев.*

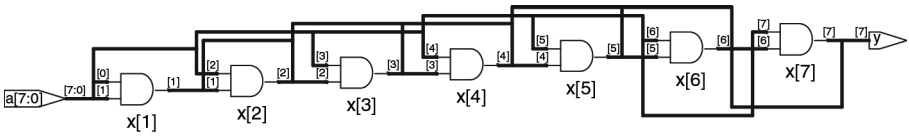


Рис. 4.30 Синтезированная схема модуля `andN`

4.9. Тестбенч

Тестбенч – это модуль на HDL, который используется для тестирования другого модуля, называемого *тестируемое устройство* (device under test, DUT)[1]. Тестбенч содержит операторы для генерации значений, по-

[1] Некоторые программы разработки называют *тестируемый модуль* (unit under test, UUT).

даваемых на входы DUT, и также для проверки, что на выходе получаются правильные значения. Наборы входных и желаемых выходных значений называются тестовыми векторами.

Проведем тестирование модуля `sillyfunction` из **раздела 4.1.1**, вычисляющего $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + ab\bar{c}$. Это простой модуль, поэтому можно проделать исчерпывающее тестирование, подавая на входы все восемь возможных тестовых векторов.

В **HDL-примере 4.37** показан простой тестбенч. Он включает в себя тестируемый блок DUT, затем подает значения векторов на его входы. Блокирующие присваивания и задержки нужны для приложения значений в желаемом порядке. Пользователь должен просмотреть результаты моделирования и проверить правильность результатов. Тестбенч моделируется так же, как и другие модули HDL, но он не является синтезируемым.

HDL-пример 4.37 ТЕСТБЕНЧ

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // задание (определение) тестируемого
  // устройства
  sillyfunction dut(a, b, c, y);

  // активировать входы пошагово,
  // с интервалом
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

Оператор `initial` выполняет содержащиеся в нем операторы в нулевой момент времени моделирования. В данном случае он подает на входы набор 000 и ждет 10 единиц времени. Затем он подает 001 и ждет еще 10 единиц времени, и так далее, пока не будут поданы все восемь возможных наборов. Операторы `initial` должны использоваться только в тестбенчах для моделирования, а не в модулях, из которых будет синтезирована аппаратура. В аппаратуре нет способа магическим образом при включении исполнять последовательности шагов.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- нет ни входов,
                     -- ни выходов
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- задание (определение) тестируемого
  -- устройства
  dut: sillyfunction port map(a, b, c, y);
  -- активировать входы пошагово, с
интервалом
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- ждать навсегда
  end process;
end;
```

Оператор `process` подает на входы набор 000 и ждет 10 нс. Затем он подает 001 и ждет еще 10 нс, и так далее, пока не будут поданы все восемь возможных наборов.

HDL-пример 4.37 (окончание)

Наконец, процесс входит в вечное ожидание, иначе его выполнение началось бы заново и он стал бы подавать тестовые векторы повторно.

Проверять правильность выходов вручную утомительно и чревато ошибками, да и тестировать в уме относительно легко, когда схема свежа в памяти. Но если придется внести в нее поправки через несколько недель, то определять впоследствии, какое значение нужно считать правильным, будет в разы труднее. Гораздо лучше разработать тестбенч с самопроверкой, показанный в **HDL-примере 4.38**.

HDL-пример 4.38 ТЕСТБЕНЧ С САМОПРОВЕРКОЙ

SystemVerilog

```
module testbench2();
  logic a, b, c, y;
  // задание (определение) тестируемого
  // устройства
  sillyfunction dut(a, b, c, y);
  // активировать входы пошагово,
  // с интервалом для проверки результатов
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1; #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("010 failed.");
    c = 1; #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1; #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("110 failed.");
    c = 1; #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

Оператор `assert` в SystemVerilog проверяет, истинно ли указанное условие. Если нет, то выполняется оператор `else`. Системная процедура `$error` в операторе `else` печатает сообщение об ошибке с указанием нарушенного условия. Операторы `assert` игнорируются при синтезе.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench2 is -- нет ни входов,
                     -- ни выходов
end;
architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- задание (определение) тестируемого
  -- устройства
  dut: sillyfunction port map(a, b, c, y);
  -- активировать входы пошагово,
  -- с интервалом для проверки результатов
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- бесконечный цикл ожидания
  end process;
end;
```

HDL-пример 4.38 (окончание)

В SystemVerilog сравнение с помощью == и != работает для сигналов, которые не принимают значения x и z. Тестбенч использует операторы === и !== для сравнений на равенство и неравенство соответственно, потому что эти операторы работают также и с операндами, значения которых могут быть x или z.

Оператор assert проверяет условие и печатает сообщение, указанное после report, если условие не выполнено. Оператор имеет смысл только при моделировании, не при синтезе.

Разрабатывать код для каждого тестового вектора тоже становится утомительно, особенно для модулей, требующих большого количества тестовых векторов. Еще лучше держать тестовые векторы в отдельном файле. Тогда тестбенч будет просто читать их из файла, подавать входной вектор на входы DUT, проверять, что значения выходов совпадают с выходным вектором, и повторять, пока не будет достигнут конец файла.

В **HDL-примере 4.39** показан такой тестбенч. Он генерирует тактовый сигнал с помощью оператора always/process без списка чувствительности, поэтому этот оператор выполняется как бесконечный цикл. В начале моделирования тестбенч читает тестовые векторы из текстового файла и устанавливает reset в течение двух тактов. Хотя тактовый сигнал и сброс не нужны для тестирования комбинационной логики, они упомянуты, потому что будут важны для тестирования последовательных устройств.

В файле example.tv находятся входы и ожидаемый выход в двоичном виде:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Новые значения входов подаются по переднему фронту тактового сигнала, а выход проверяется по заднему фронту. Сообщения об ошибках выдаются в момент возникновения ошибок. В конце моделирования тестбенч выводит итоговое сообщение в консоль отладки: общее количество тестовых векторов и количество обнаруженных ошибок.

Среда в **HDL-примере 4.39** избыточна для такой простой схемы. Но ее легко изменить для тестирования более сложных схем, заменив файл example.tv, включив в среду другое тестируемое устройство и изменив несколько строк кода для установки входов и проверки выходов.

HDL-пример 4.39 ТЕСТБЕНЧ С ФАЙЛОМ ТЕСТОВЫХ ВЕКТОРОВ

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // задание (определение) тестируемого
  // устройства
  sillyfunction dut(a, b, c, y);

  // генерировать такты
  always
   begin
     clk = 1; #5; clk = 0; #5;
   end

  // при старте теста загрузить векторы
  // и дать импульс сброса
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // подать тестовые векторы по переднему
  // фронту такта
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} =
testvectors[vectornum];
    end

  // проверить результаты по заднему
  // фронту такта
  always @(negedge clk)
   if (~reset) begin // пропустить проверку
                     // при сбросе
     // проверить результаты
     if (y !== yexpected) begin
         $display("Error: inputs = %b", {a,
b, c});
        $display(" outputs = %b (%b
expected)", y, yexpected);
        errors = errors + 1;
     end
     vectornum = vectornum + 1;
     if (testvectors[vectornum] === 4'bx)
begin
       $display("%d tests completed with %d
errors", vectornum, errors);
       $finish;
     end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- нет ни входов,
                     -- ни выходов
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- задание (определение) тестируемого
  -- устройства
  dut: sillyfunction port map(a, b, c, y);

  -- генерировать такты
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- при старте теста дать импульс сброса
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- запустить тест
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector
(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop

      -- менять векторы по переднему
      -- фронту
      wait until rising_edge(clk);

      -- читать следующую строку тест-векторов
      -- и разделить ее на части
      readline(tv, L);
      read(L, vector_in);
```

HDL-пример 4.39. (окончание)

`$readmemb` читает файл с двоичными числами в массив `testvectors`. `$readmemh` работает аналогично, но читает файл с шестнадцатеричными числами.

Следующий блок кода ждет одну единицу времени после переднего фронта тактового сигнала (чтобы избежать путаницы, если тактовый сигнал и данные меняются одновременно), затем устанавливает три входа (`a`, `b` и `c`) и ожидаемый выход (`yexpected`) в соответствие с четырьмя битами в текущем тестовом векторе.

Среда сравнивает полученный выход, `y`, с ожидаемым выходом, `yexpected`, и печатает сообщение об ошибке, если они не совпадают. `%b` и `%d` означают печать значений в двоичном и десятичном виде соответственно. `$display` – это системная процедура печати в консоль среды моделирования. Например, `$display("%b %b", y, yexpected);` печатает два значения, `y` и `yexpected`, в двоичном виде. `%h` печатает в шестнадцатеричном виде.

Этот процесс повторяется, пока в массиве `testvectors` не закончатся прочитанные из файла тестовые векторы. `$finish` завершает моделирование.

Обратите внимание, что хотя модуль на SystemVerilog предусматривает вплоть до 10 001 тестового вектора, моделирование завершится после подачи восьми векторов из файла.

```
      read(L, dummy); -- обойти нижнее
                         подчеркивание
      read(L, vector_out);
      (a, b, c) <= vector_in(2 downto 0)
after 1 ns;
      y_expected <= vector_out after 1 ns;

      -- -- проверить результаты
      -- -- по заднему фронту
      wait until falling_edge(clk);
      if y /= y_expected then
         report "Error: y = " & std_
logic'image(y);
         errors := errors + 1;
       end if;
       vectornum := vectornum + 1;
     end loop;

     -- собрать результаты в конце
     -- моделирования
     if (errors = 0) then
      report "NO ERRORS -- " &
             integer'image(vectornum) &
             " tests completed successfully."
             severity failure;
    else
      report integer'image(vectornum) &
             " tests completed, errors = " &
             integer'image(errors)
             severity failure;
    end if;
  end process;
end;
```

Код на VHDL использует команды чтения из файла, рассмотрение которых не входит в эту главу, но дает понимание, как выглядит тестбенч с самопроверкой на VHDL.

4.10. Заключение

Языки описания аппаратуры (HDL) – очень важные инструменты разработчиков современной цифровой электроники. Изучив SystemVerilog или VHDL, вы сможете разрабатывать цифровые системы гораздо быстрее, чем при традиционном черчении принципиальных схем. Цикл отладки тоже обычно гораздо короче, так как изменения заключаются в редактировании текста, а не утомительном переподключении проводов на схеме. Но с использованием HDL цикл отладки может быть и гораздо

дольше, если вы плохо представляете себе, какую аппаратуру описывает ваш код.

Языки описания аппаратуры используются и для моделирования, и для синтеза. Моделирование – мощный способ протестировать систему на компьютере, перед тем как она превратится в аппаратуру. Среда моделирования позволяет проверить те значения сигналов в системе, которые могут быть недоступны для измерения на реальной электрической схеме. Логический синтез превращает код на HDL в цифровые логические схемы.

Самое важное, что вам нужно помнить при разработке кода на HDL, – это то, что вы описываете настоящую аппаратуру, а не разрабатываете программу для компьютера. Начинающие разработчики часто совершают ошибку, создавая код на HDL, не продумав, какую именно аппаратуру они хотят получить. Если вы не знаете, какая аппаратура получится в результате синтеза из кода, вы, скорее всего, не достигнете нужного результата. Поэтому начинайте с эскиза блочной диаграммы системы, определяя, какие ее части являются комбинационной логикой, а какие – последовательностными схемами или конечными автоматами и т. д. Затем ведите разработку для каждой части на HDL, используя правильные конструкции для нужного типа аппаратуры.

Упражнения

Упражнения в этом разделе можно выполнять на языке, который вам больше нравится. Если у вас есть приложение для моделирования, протестируйте то, что вы создали. Выведите значения сигналов и объясните, как они доказывают, что схема работает правильно. Если у вас есть синтезатор, синтезируйте схему. Напечатайте полученную принципиальную схему и объясните, почему она удовлетворяет ожиданиям.

Упражнение 4.1 Нарисуйте диаграмму схемы, описанной программой ниже. Упростите схему, добившись минимума логических элементов.

SystemVerilog

```
module exercise1(input logic a, b, c,
                 output logic y, z);

  assign y = a & b & c |a & b & ~c | a
& ~b & c;
  assign z = a & b |~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or
       (a and not b and c);
  z <= (a and b) or (not a and not b);
```

Упражнение 4.2 Нарисуйте диаграмму схемы, описанной программой ниже. Упростите схему, добившись минимума логических элементов.

SystemVerilog

```
module exercise2(input  logic[3:0] a,
                 output logic [1:0] y);

  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(all) begin
    if    a(0) then y <= "11";
    elsif a(1) then y <= "10";
    elsif a(2) then y <= "01";
    elsif a(3) then y <= "00";
    else            y <= a(1 downto 0);
    end if;
end process;
```

Упражнение 4.3 Разработайте модуль на HDL, вычисляющий четырехвходовую функцию XOR (исключающее ИЛИ). Вход обозначьте $a_{3:0}$, выход – y.

Упражнение 4.4 Разработайте тестбенч с самопроверкой для **упражнения 4.3**. Создайте файл, содержащий все 16 вариантов входов. Проведите моделирование схемы схему и убедитесь, что она работает. Внесите ошибку в файл с тестовыми векторами и убедитесь, что тестбенч сообщает о несовпадении результатов.

Упражнение 4.5 Разработайте на HDL модуль `minority` с тремя входами, `a`, `b`, и `c`, и одним выходом, `y`, принимающим значение TRUE, если не менее двух входов равны FALSE.

Упражнение 4.6 Разработайте на HDL модуль для управления семисегментным индикатором шестнадцатеричных цифр. Должны поддерживаться не только цифры 0–9, но и A, B, C, D, E и F.

Упражнение 4.7 Разработайте тестбенч с самопроверкой для **упражнения 4.6**. Создайте файл, содержащий все 16 вариантов входов. Проведите моделирование схемы схему и убедитесь, что она работает. Внесите ошибку в файл с тестовыми векторами и убедитесь, что тестбенч сообщает о несовпадении результатов.

Упражнение 4.8 Разработайте восьмивходовый мультиплексор с именем `mux8`, входами $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7` и выходом `y`.

Упражнение 4.9 Разработайте структурный модуль для вычисления логической функции $y = a\bar{b} + \bar{b}\,\bar{c} + \bar{a}bc$ с помощью построения логических схем на мультиплексорах. Используйте мультиплексор из **упражнения 4.8**.

Упражнение 4.10 Повторите **упражнение 4.9** с помощью четырехвходового мультиплексора и любого количества логических элементов НЕ.

Упражнение 4.11 В **разделе 4.5.4** было отмечено, что синхронизатор можно описать с помощью блокирующих присваиваний в правильном порядке. Придумайте простую последовательностную схему, которую нельзя правильно описать с помощью блокирующих присваиваний, независимо от их порядка.

Упражнение 4.12 Разработайте модуль на HDL для схемы приоритетов с восемью входами.

Упражнение 4.13 Разработайте модуль на HDL для дешифратора 2:4.

Упражнение 4.14 Разработайте модуль на HDL для дешифратора 6:64 с помощью трех экземпляров дешифратора 2:4 из **упражнения 4.13** и нескольких трехвходовых логических элементов И.

Упражнение 4.15 Разработайте модуль на HDL, реализующий логические выражения из **упражнения 2.13**.

Упражнение 4.16 Разработайте модуль на HDL, реализующий схему из **упражнения 2.26**.

Упражнение 4.17 Разработайте модуль на HDL, реализующий схему из **упражнения 2.27**.

Упражнение 4.18 Разработайте модуль на HDL, реализующий логическую функцию из **упражнения 2.28**. Обратите особое внимание на то, как обходиться с незначащими битами.

Упражнение 4.19 Разработайте модуль на HDL, реализующий функции из **упражнения 2.35**.

Упражнение 4.20 Разработайте модуль на HDL, реализующий кодер с приоритетами из **упражнения 2.36**.

Упражнение 4.21 Разработайте модуль на HDL, реализующий модифицированный кодер с приоритетами из **упражнения 2.37**.

Упражнение 4.22 Разработайте модуль на HDL, реализующий преобразователь из бинарного в унарный код из **упражнения 2.38**.

Упражнение 4.23 Разработайте модуль на HDL, который реализует функцию проверки количества дней в месяце из **вопроса 2.2**.

Упражнение 4.24 Нарисуйте диаграмму состояний конечного автомата, описанного кодом на HDL, приведенным ниже:

SystemVerilog

```
module fsm2(input  logicclk, reset,
            input  logica, b,
            output logicy);

  logic [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
       S0: if (a ^ b) nextstate = S1;
           else       nextstate = S0;
       S1: if (a & b) nextstate = S2;
           else       nextstate = S0;
       S2: if (a | b) nextstate = S3;
           else       nextstate = S0;
       S3: if (a | b) nextstate = S3;
           else       nextstate = S0;
    endcase
  assign y = (state == S1) |(state ==
S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 =>if (a xor b) then
                      nextstate <= S1;
                else nextstate <= S0;
                end if;
      when S1 =>if (a and b) then
                      nextstate <= S2;
                else nextstate <= S0;
                end if;
      when S2 =>if (a or b) then
                      nextstate <= S3;
                else nextstate <= S0;
                end if;
      when S3 =>if (a or b) then
                      nextstate <= S3;
                else nextstate <= S0;
                end if;

     end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

Упражнение 4.25 Нарисуйте диаграмму состояний конечного автомата, описанного кодом на HDL, привеженным ниже. Автоматы подобного типа используются для предсказания переходов в некоторых микропроцессорах.

SystemVerilog

```
module fsm1(input logic clk, reset,
            input logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter S0 = 5'b00001;
  parameter S1 = 5'b00010;
  parameter S2 = 5'b00100;
  parameter S3 = 5'b01000;
  parameter S4 = 5'b10000;
  always_ff @(posedge clk, posedge reset)
   if (reset) state <= S2;
   else       state <= nextstate;
  always_comb
    case (state)
      S0: if (taken) nextstate = S1;
          else       nextstate = S0;
      S1: if (taken) nextstate = S2;
          else       nextstate = S0;
      S2: if (taken) nextstate = S3;
          else       nextstate = S1;
      S3: if (taken) nextstate = S4;
          else       nextstate = S2;
      S4: if (taken) nextstate = S4;
          else       nextstate = S3;
      default:       nextstate = S2;
    endcase
  assign predicttaken = (state == S4) |
                        (state == S3) |
                        (state == S2 &&
back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164. all;
entity fsm1 is
  port(clk, reset:    in  STD_LOGIC;
       taken, back:   in  STD_LOGIC;
       predicttaken:  out STD_LOGIC);
end;
architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;
process(all) begin
   case state is
    when S0 => if taken then
                    nextstate <= S1;
               else nextstate <= S0;
               end if;
    when S1 => if taken then
                    nextstate => S2;
               else nextstate <= S0;
               end if;
    when S2 => if taken then
                    nextstate <= S3;
               else nextstate <= S1;
               end if;
    when S3 => if taken then
                    nextstate <= S4;
               else nextstate <= S2;
               end if;
    when S4 => if taken then
                    nextstate <= S4;
               else nextstate <= S3;
               end if;
    when others => nextstate <= S2;
  end case;
end process;
-- логика выхода
predicttaken <= '1' when
               ((state = S4) or (state = S3) or
               (state = S2 and back = '1'))
  else '0';
end;
```

Упражнение 4.26 Разработайте модуль на HDL для SR-защелки.

Упражнение 4.27 Разработайте модуль на HDL для JK-триггера со входами *clk*, *J* и *K* и выходом *Q*. По переднему фронту тактового сигнала *Q* сохраняет предыдущее состояние, если $J = K = 0$, становится равным 1, если $J = 1$, сбрасывается в 0, если $K = 1$, и инвертируется, если $J = K = 1$.

Упражнение 4.28 Разработайте модуль на HDL для защелки на **рис. 3.18**. Используйте один оператор присваивания для каждого логического элемента. Задайте задержку 1 (или 1 нс) для каждого логического элемента. Проведите моделирование защелки защелку и убедитесь, что она работает правильно. Затем увеличьте задержку у инвертора. Насколько большой может быть задержка, прежде чем защелка перестанет работать корректно из-за гонки сигналов?

Упражнение 4.29 Разработайте модуль на HDL для контроллера светофора из **раздела 4.3.1**.

Упражнение 4.30 Разработайте три модуля на HDL для параметризированного контроллера светофора с режимом парада из **примера 3.8**. Назовите эти модули `controller`, `mode` и `lights` и назовите их входы-выходы как на **рис. 3.33 (b)**.

Упражнение 4.31 Разработайте модуль на HDL, описывающий схему на **рис. 3.42**.

Упражнение 4.32 Разработайте модуль на HDL для конечного автомата с диаграммой состояний, изображенной на **рис. 3.69** из **упражнения 3.22**.

Упражнение 4.33 Разработайте модуль на HDL для конечного автомата с диаграммой состояний, изображенной на **рис. 3.70** из **упражнения 3.23**.

Упражнение 4.34 Разработайте модуль на HDL для улучшенного контроллера светофора из **упражнения 3.24**.

Упражнение 4.35 Разработайте модуль на HDL для дочки-улитки из **упражнения 3.25**.

Упражнение 4.36 Разработайте модуль на HDL для дозатора напитков из **упражнения 3.26**.

Упражнение 4.37 Разработайте модуль на HDL для счетчика в коде Грея из **упражнения 3.27**.

Упражнение 4.38 Разработайте модуль на HDL для счетчика в коде Грея ВВЕРХ/ВНИЗ из **упражнения 3.28**.

Упражнение 4.39 Разработайте модуль на HDL для конечного автомата из **упражнения 3.29**.

Упражнение 4.40 Разработайте модуль на HDL для конечного автомата из **упражнения 3.30**.

Упражнение 4.41 Разработайте модуль на HDL для последовательного вычисления противоположного значения из **вопроса 3.2**.

Упражнение 4.42 Разработайте модуль на HDL для схемы из **упражнения 3.31**.

Упражнение 4.43 Разработайте модуль на HDL для схемы из **упражнения 3.32**.

Упражнение 4.44 Разработайте модуль на HDL для схемы из **упражнения 3.33**.

Упражнение 4.45 Разработайте модуль на HDL для схемы из **упражнения 3.34**, при желании с использованием полного сумматора из **раздела 4.2.5**.

Упражнения для SystemVerilog

Упражнение 4.46 Что значит, когда в SystemVerilog сигнал объявлен как `tri`?

Упражнение 4.47 Переработайте модуль `syncbad` из **примера 4.29**. Используйте неблокирующие присваивания, но измените код так, чтобы получился правильный синхронизатор с двумя триггерами.

Упражнение 4.48 Рассмотрите следующие два модуля на SystemVerilog. Функционально одинаковы ли они? Нарисуйте схему аппаратуры реализующей каждый из них.

```
module code1(input  logic clk, a, b, c,
             output logic y);

  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2 (input logic a, b, c, clk,
              output logic y);
  logic x;
  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

Упражнение 4.49 Повторите **упражнение 4.48**, если в каждом присваивании <= заменено на =.

Упражнение 4.50 В приведенных ниже модулях на SystemVerilog показаны типичные ошибки, замеченные авторами у студентов при выполнении лабораторных работ. Объясните ошибку в каждом модуле и укажите, как ее исправить.

(a)
```
module latch(input logic      clk,
             input logic[3:0] d,
             output reg [3:0] q);

  always @(clk)
    if (clk) q <= d;
endmodule
```

(b)
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

  always @(a)
```

```
    begin
      y1 = a & b;
      y2 = a |b;
      y3 = a ^ b;
      y4 = ~(a & b);
      y5 = ~(a |b);
    end
endmodule
```

(c)

```
module mux2(input  logic [3:0] d0, d1,
            input  logic   s,
            output logic [3:0] y);

  always @(posedge s)
    if (s) y <= d1;
    else y <= d0;
endmodule
```

(d)

```
module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);

  always @(posedge clk)
    q1 = d1;
    q0 = d0;
endmodule
```

(e)

```
module FSM(input  logic clk,
           input  logic a,
           output logic out1, out2);

  logic state;

  // логика и регистр следующего состояния (последовательностная)
  always_ff @(posedge clk)
    if (state == 0) begin
        if (a) state <= 1;
    end else begin
        if (~a) state <= 0;
    end
  always_comb // логика выхода (комбинационная)
    if (state == 0) out1 = 1;
    else            out2 = 1;
endmodule
```

(f)

```
module priority(input  logic [3:0] a,
                output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
endmodule
```

(g)

```
module divideby3FSM(input  logicclk,
                    input  logicreset,
                    output logicout);

  logic [1:0] state, nextstate;
```

```
  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  // регистр состояния
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // логика следующего состояния
  always @(state)
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
    endcase

  // логика выхода
  assign out = (state == S2);
endmodule
```

(h)
```
module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output tri [3:0] y);
  tristate t0(d0, s, y);
  tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
                input  logic [3:0] d,
                output logic [3:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;

  always @(set)
    if (set)  q <= 1;
endmodule
```

(j)
```
module and3(input  logic a, b, c,
            output logic y);

  logic tmp;

  always @(a, b, c)
  begin
    tmp <= a & b;
    y <= tmp & c;
  end
endmodule
```

Упражнения для VHDL

Упражнение 4.51 Зачем в VHDL надо писать

```
q <= '1' when state = S0 else '0';
```

а не просто

```
q <= (state = S0);
```

Упражнение 4.52 В каждом из нижеследующих модулей на VHDL есть ошибка. Для краткости показаны лишь описания архитектуры; считайте, что объявление библиотеки и объявление интерфейса правильные.

Объясните ошибку и опишите, как ее исправить.

(a)
```
architecture synth of latch is
begin
 process(clk) begin
   if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process(a) begin
    Y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process(clk)
    if rising_edge(clk) then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process(all) begin
    if  a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    end if;
  end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;
```

```
    process(state) begin
      case state is
        when S0 =>nextstate <= S1;
        when S1 =>nextstate <= S2;
        when S2 =>nextstate <= S0;
      end case;
    end process;
    q <= '1' when state = S0 else '0';
  end;
```

(f)
```
  architecture struct of mux2 is
    component tristate
      port(a: in   STD_LOGIC_VECTOR(3 downto 0);
           en: in  STD_LOGIC;
           y:  out STD_LOGIC_VECTOR(3 downto 0));
    end component;
  begin
    t0: tristate port map(d0, s, y);
    t1: tristate port map(d1, s, y);
  end;
```

(g)
```
  architecture asynchronous of floprs is
  begin
    process(clk, reset) begin
      if reset then
        q <= '0';
      elsif rising_edge(clk) then
        q <= d;
      end if;
    end process;
    process(set) begin
      if set then
        q <= '1';
      end if;
    end process;
  end;
```

Вопросы для собеседования

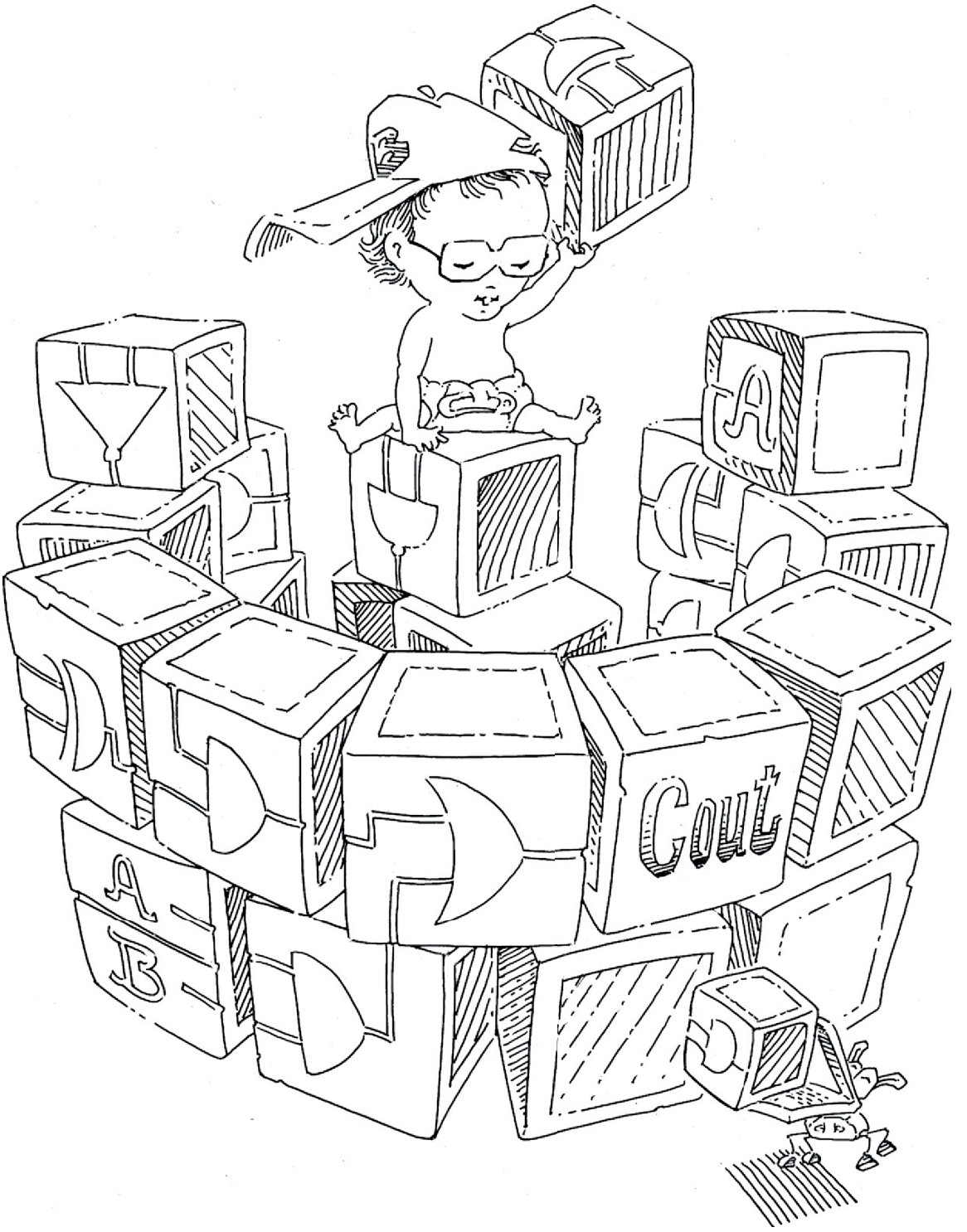
Эти вопросы задавались на собеседованиях по приему на работу, связанную с разработкой цифровых систем.

Вопрос 4.1 Напишите код на HDL, реализующую управление 32-битной шиной `data` сигналом `sel`, получая 32-битный сигнал `result`. Если `sel` истинно, `result` = `data`, иначе все биты `result` – нули.

Вопрос 4.2 Объясните разницу между блокирующими и неблокирующими присваиваниями в SystemVerilog. Приведите примеры.

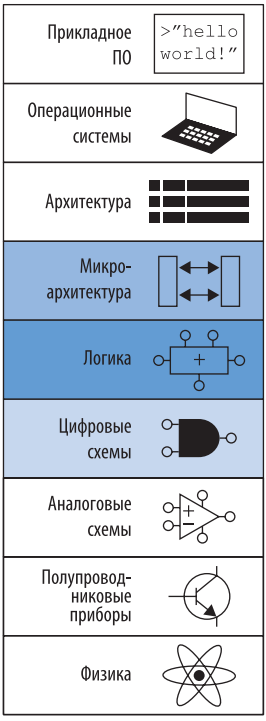
Вопрос 4.3 Что делает этот оператор SystemVerilog:

```
result = | (data[15:0] & 16'hC820);
```


A
Cout
A
B

Глава 5

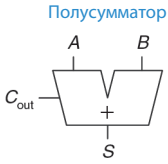
Цифровые функциональные узлы



5.1. Введение

В предыдущих главах мы познакомились с разработкой комбинационных и последовательностных схем с использованием логических выражений, схем и языков описания аппаратуры. В этой главе мы рассмотрим более сложные комбинационные и последовательностные функциональные узлы, используемые в цифровых системах. Такие узлы включают в себя арифметические схемы, счетчики, схемы сдвига, матрицы памяти и матрицы логических элементов. Эти функциональные узлы полезны не только сами по себе, но и как демонстрация принципов иерархичности, модульности и регулярности. Функциональные узлы иерархически собраны из нескольких простейших компонент, таких как логические

элементы, мультиплексоры и дешифраторы. Каждый функциональный узел имеет четко определенный интерфейс и может рассматриваться как черный ящик, когда не важна его базовая реализация. Регулярная структура каждого функционального узла может расширяться до любого размера. В **главе 7** подобные функциональные узлы будут использоваться для создания микропроцессора.



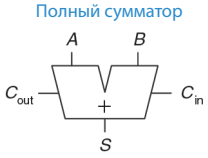
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

Рис. 5.1 Одноразрядный полусумматор

```
  1
 0001
+0101
 0110
```

Рис. 5.2 Бит переноса



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Рис. 5.3 Одноразрядный полный сумматор

5.2. Арифметические схемы

Арифметические схемы являются основным функциональным узлом любого компьютера. Компьютеры и цифровые схемы выполняют множество арифметических операций: сложение, вычитание, сравнение, сдвиги, умножение и деление. В этой главе будет описана аппаратная реализация всех перечисленных операций.

5.2.1. Сложение

Сложение – одна из самых распространенных операций в цифровых системах. Для начала мы рассмотрим сложение двух одноразрядных двоичных чисел. Затем мы расширим эту процедуру до N-разрядных чисел. Сумматоры демонстрируют компромисс между скоростью и сложностью реализации.

Полусумматор

Вначале разработаем одноразрядный *полусумматор* (*half adder*). Как показано на **рис. 5.1**, полусумматор имеет два входа (A и B) и два выхода (S и C_{out}). S – это сумма A и B. Если и A, и B равны 1, то выход S должен стать равным 2, такое число не может быть представлено в виде одного двоичного разряда. В этом случае результат указывается вместе с переносом C_{out} в следующий разряд. Полусумматор может быть построен из элементов XOR (Исключающее ИЛИ) и AND (логическое И).

В многоразрядном сумматоре выход C_{out} подсоединяется к входу переноса следующего разряда. Например, на **рис. 5.2** бит переноса показан синим цветом, он является выходом C_{out} одноразрядного сумматора 1-го разряда и входом C_{in} сумматора следующего разряда. При этом в полусумматоре нет входа переноса C_{in} для связи с выходом C_{out} предыдущего разряда. В *полном сумматоре*, рассматриваемом в следующем разделе, такой вход есть.

Полный сумматор

Как показано на **рис. 5.3**, *полный сумматор* (*full adder*), описанный в **разделе 2.1**, имеет вход переноса C_{in}. На рисунке также приведены уравнения для S и C_{out}.

Сумматор с распространяющимся переносом

N-разрядный сумматор складывает 2 N-разрядных числа (A и B), а также входной перенос C_{in} и формирует N-разрядный результат S и выходной перенос C_{out}. Такой сумматор называется *сумматором с распространяющимся переносом* (*carry propagate adder, CPA*), так как выходной перенос одного разряда переходит в следующий разряд. Условное обозначение такого сумматора показано на **рис. 5.4**. Оно аналогично обозначению полного сумматора, за исключением того, что входы / выходы A, B, S являются шинами, а не отдельными разрядами. Самыми распространенными реализациями CPA являются: *сумматоры с последовательным переносом* (ripple-carry adders), *с ускоренным переносом* (carry-lookahead adders) и *префиксные сумматоры* (prefix adders).

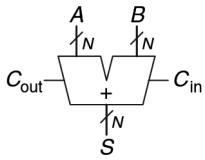


Рис. 5.4 Сумматор с распространяющимся переносом

Сумматоры с последовательным переносом

Самый простой способ реализации N-разрядного сумматора – это объединение в цепь N полных сумматоров. Выход C_{out} некоторого разряда будет поступать на вход C_{in} следующего разряда и т. д. (**рис. 5.5**).

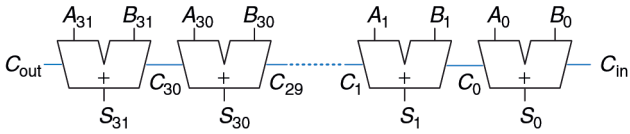


Рис. 5.5 32-разрядный сумматор с последовательным переносом

Такая схема называется *сумматором с последовательным переносом* (*ripple-carry adder*). При ее разработке используется принцип модульности и регулярности: модуль полного сумматора многократно используется для формирования большей схемы. Такой сумматор имеет недостаток: его скорость падает при увеличении количества разрядов N. S_{31} зависит от C_{30}, который зависит от C_{29}, который, в свою очередь, зависит от C_{28} и т. д. до C_{in} (**рис. 5.5**). Перенос проходит через всю цепь. Задержка такого сумматора (t_{ripple}) увеличивается вместе с количеством разрядов, как показано в **уравнении (5.1)**, где t_{FA} – это задержка полного сумматора.

$$t_{ripple} = Nt_{FA}. \tag{5.1}$$

Сумматоры с ускоренным переносом

Главной причиной того, что большие сумматоры с последовательным переносом работают медленно, является то, что сигнал переноса должен пройти через все биты сумматора. *Сумматоры с ускоренным переносом* (*carry-lookahead* adder, *CLA*) – это другой тип суммато-

ров с распространяющимся переносом, который решает эту проблему путем разделения сумматора на *блоки* и реализации схемы так, чтобы определить выходной перенос блока, как только стал известен его входной перенос. Таким образом, мы смотрим вперед через блоки и не ждем прохождения переноса через все полные сумматоры внутри блока. К примеру, 32-разрядный сумматор может быть разделен на восемь 4-разрядных сумматоров.

Сумматоры с ускоренным переносом используют сигналы *генерации* (G) и *распространения* (P), которые описывают, как блок (или разряд) определяет выход переноса. i-й разряд сумматора генерирует перенос, если он выдает перенос на своем выходе, независимо от наличия переноса на входе. i-й разряд сумматора генерирует C_i в том случае, если и A_i, и B_i равны 1. Таким образом, сигнал генерации G_i можно вычислить как $G_i = A_iB_i$. Разряд называется *распространяющим*, если выходной сигнал переноса появляется при наличии входного переноса. Разряд будет распространять входной сигнал переноса, C_{i-1}, если либо A_i, либо B_i равны 1. Таким образом, $P_i = A_i + B_i$. Используя эти определения, мы можем описать логику формирования сигнала переноса для определенного разряда. Разряд i сумматора будет формировать выходной сигнал переноса C_i, если он или генерирует перенос G_i, или распространяет входной перенос P_iC_{i-1}. В виде уравнения это можно записать следующим образом:

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1}. \quad (5.2)$$

Определения сигналов генерации и распространения относятся и к многоразрядным блокам. Блок называется генерирующим перенос, если он создает выходной перенос независимо от входного сигнала переноса данного блока. Блок называется распространяющим перенос, если выходной перенос возникает при поступлении входного переноса. $G_{i:j}$ и $P_{i:j}$ определяются как сигналы генерации и распространения для блоков, соответствующих разрядам разряды с i до j.

Блок генерирует перенос, если самый старший разряд генерирует перенос или если старший разряд распространяет перенос, сгенерированный предыдущим разрядом и т. д. Например, логика блока генерации для блока, охватывающего разряды от 0 до 3, будет следующей:

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)). \quad (5.3)$$

Обычно в электронных схемах сигналы распространяются слева направо. Арифметические схемы нарушают эти правила, так как перенос идет справа налево (от младшего разряда к старшему).

В течение многих лет люди используют множество способов для выполнения арифметических действий. Дети считают на пальцах (и некоторые взрослые, кстати, тоже). Китайцы и вавилоняне изобрели счеты еще в 2400 г. до н. э. Логарифмические линейки, придуманные в 1630 году, использовались вплоть до 1970-х, затем стали входить в обиход ручные инженерные калькуляторы. Сегодня компьютеры и цифровые калькуляторы используются повсеместно. Что придумают дальше?

Блок распространяет перенос, если все входящие в него разряды этот перенос распространяют. Логика распространения для блока, соответствующего разрядам с 0 до 3:

$$P_{3:0} = P_3 P_2 P_1 P_0. \tag{5.4}$$

При помощи блоковых сигналов генерации и распространения можно быстро определить выходной перенос блока C_i, используя его входной перенос C_{j-1}:

$$C_i = G_{i:j} + P_{i:j} C_{j-1}. \tag{5.5}$$

На **рис. 5.6 (a)** изображен 32-разрядный сумматор с ускоренным переносом, состоящий из восьми 4-разрядных блоков. Каждый блок содержит 4-разрядный сумматор с последовательным переносом и схему ускоренного переноса, определяющую выходной перенос блока по входному, которая показана на **рис. 5.6 (b)**. На рисунке не представлены элементы И и ИЛИ, необходимые для вычисления одноразрядных сигналов генерации и распространения G_i и P_i по A_i и B_i. Сумматор с ускоренным переносом демонстрирует применение принципов модульности и регулярности.

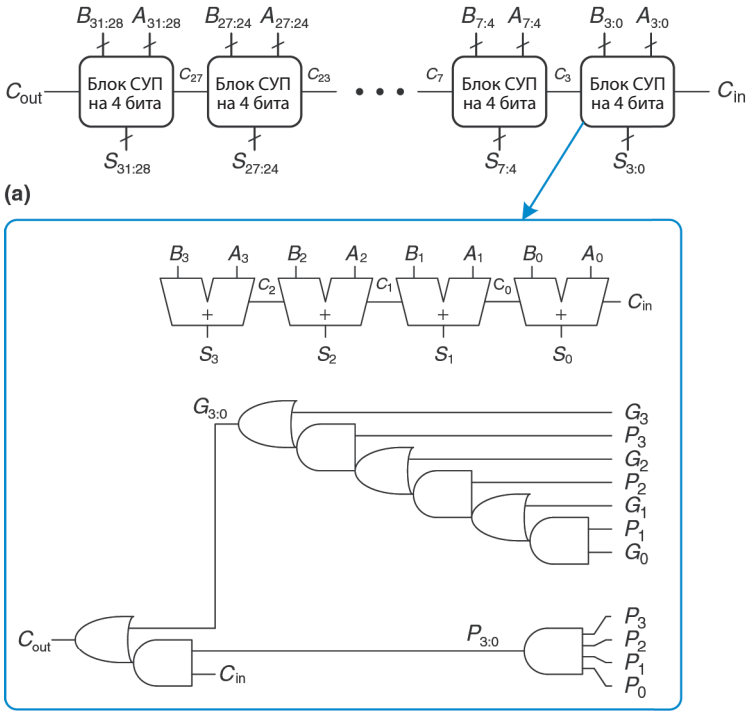


Рис. 5.6 (a) 32-разрядный сумматор с ускоренным переносом и (b) его 4-битный блок

Все блоки сумматора одновременно вычисляют однобитные и блоковые сигналы генерации и распространения. Критический путь начинается с вычисления G_0 и $G_{3:0}$ в первом блоке сумматора. Сигнал C_{in} затем распространяется по направлению к C_{out} через логические элементы И/ИЛИ всех блоков. Для большого сумматора это происходит гораздо быстрее, чем распространение переноса через каждый последующий разряд сумматора. И наконец, критический путь через последний блок содержит небольшой сумматор с последовательным переносом. Таким образом, N-разрядный сумматор, разделенный на k-разрядные блоки, имеет задержку

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k-1)t_{\text{AND\_OR}} + k_{tFA}, \tag{5.6}$$

где t_{pg} – задержка отдельных логических элементов генерации/распространения (одиночных логических элементов И/ИЛИ) при генерации P и G. t_{pg_block} является задержкой формирования сигналов генерации/распространения $P_{i:j}$ и $G_{i:j}$ для k-разрядного блока, а $t_{\text{AND\_OR}}$ является задержкой тракта C_{in} – C_{out}, в который входит логика И/ИЛИ k-разрядного CLA-блока. При $N > 16$ такой сумматор работает гораздо быстрее, чем сумматор с последовательным переносом. При этом задержка сумматора по-прежнему линейно возрастает с ростом N.

Пример 5.1 ЗАДЕРЖКИ СУММАТОРОВ С ПОСЛЕДОВАТЕЛЬНЫМ И УСКОРЕННЫМ ПЕРЕНОСАМИ

Сравним задержки 32-разрядного сумматора с последовательным переносом и 32-разрядного сумматора с ускоренным переносом, который состоит из 4-разрядных блоков. Предположим, что задержка каждого двухвходового логического элемента составляет 100 пс, а задержка полного сумматора – 300 пс.

Решение В соответствии с **формулой (5.1)** задержка распространения 32-разрядного сумматора с последовательным переносом равна 32×300 пс = 9.6 нс.

У сумматора с ускоренным переносом t_{pg} = 100 пс, $t_{pg\_\text{block}} = 6 \times 100$ пс = 600 пс и $t_{\text{AND\_OR}} = 2 \times 100$ пс = 200 пс. В соответствии с **уравнением (5.6)** задержка распространения 32-разрядного сумматора с ускоренным переносом, состоящего из 4-разрядных блоков, равна 100 пс + 600 пс + $(32/4 - 1) \times 200$ пс + (4×300) пс = 3.3 нс, что почти в три раза меньше, чем у сумматора с последовательным переносом.

Префиксный сумматор

Префиксный сумматор развивает идею генерации и распространения сумматора с ускоренным переносом для еще более быстрого выполнения операции сложения. Сначала он вычисляет G и P для пар разрядов, далее для блоков из четырех разрядов, затем для блоков из 8, 16 и т. д. разрядов, пока сигнал генерации не будет известен для каждого разряда. Сумма определяется всеми сигналами генерации.

То есть стратегия префиксного сумматора заключается в вычислении входного сигнала переноса C_{i-1} для каждого разряда так быстро, насколько это возможно. Затем по формуле вычисляется сумма:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}. \tag{5.7}$$

Определим разряд $i = -1$ для вычисления C_{in}: $G_{-1} = C_{in}$ и $P_{-1} = 0$. Следовательно, $C_{i-1} = G_{i-1:-1}$, так как выходной сигнал переноса $(i-1)$-го разряда будет активным, если блок, охватывающий разряды от $i-1$ до -1, генерирует перенос. Полученный перенос генерируется или в разряде $(i-1)$, или в предыдущем разряде и затем распространяется дальше. Следовательно, мы можем переписать уравнение **(5.7)** в таком виде:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}. \tag{5.8}$$

Таким образом, основной проблемой является быстрое вычисление всех блоковых сигналов генерации $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, ..., $G_{N-2:-1}$. Эти сигналы вместе с $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, ..., $P_{N-2:-1}$ называют *префиксными*.

На **рис. 5.7** показан 16-разрядный префиксный сумматор. Его работа начинается с предварительного формирования сигналов P_i и G_i для всех разрядов A_i и B_i с использованием элементов И и ИЛИ. Затем используется $\log_2 N = 4$ уровня черных ячеек для формирования префиксов $G_{i:j}$ и $P_{i:j}$. Черная ячейка принимает входы из верхней части блока, соответствующего битам $i{:}k$, и из нижней части блока, соответствующего битам $k{-}1{:}j$. Затем эти части объединяются для формирования сигналов генерации и распространения всего блока, соответствующего битам $i{:}j$. Используя уравнения **(5.9)** и **(5.10)**, получим

Первые компьютеры использовали сумматоры с ускоренным переносом, так как компоненты стоили очень дорого, а такие сумматоры используют меньше аппаратных ресурсов. Практически все современные компьютеры используют префиксные сумматоры в критических путях, так как транзисторы стали дешевле, а быстродействие — один из важнейших показателей.

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}; \tag{5.9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}. \tag{5.10}$$

Другими словами, блок, соответствующего битам $i{:}j$, будет генерировать сигнал переноса, если верхняя часть генерирует перенос или если она распространяет перенос, сгенерированный в нижней части. Блок будет распространять перенос, если и верхняя, и нижняя части распространяют его. В итоге префиксный сумматор вычисляет сумму на основе уравнения **(5.8)**.

Таким образом, задержка префиксного сумматора достигает значения, которое возрастает с увеличением количества разрядов сумматора логарифмически, а не линейно. Ускорение значительное, особенно для сумматоров, имеющих 32 и более разрядов. Такой сумматор использует существенно больше аппаратных средств, чем простой сумматор с ускоренным переносом. Сеть черных ячеек называется *префиксным деревом*.

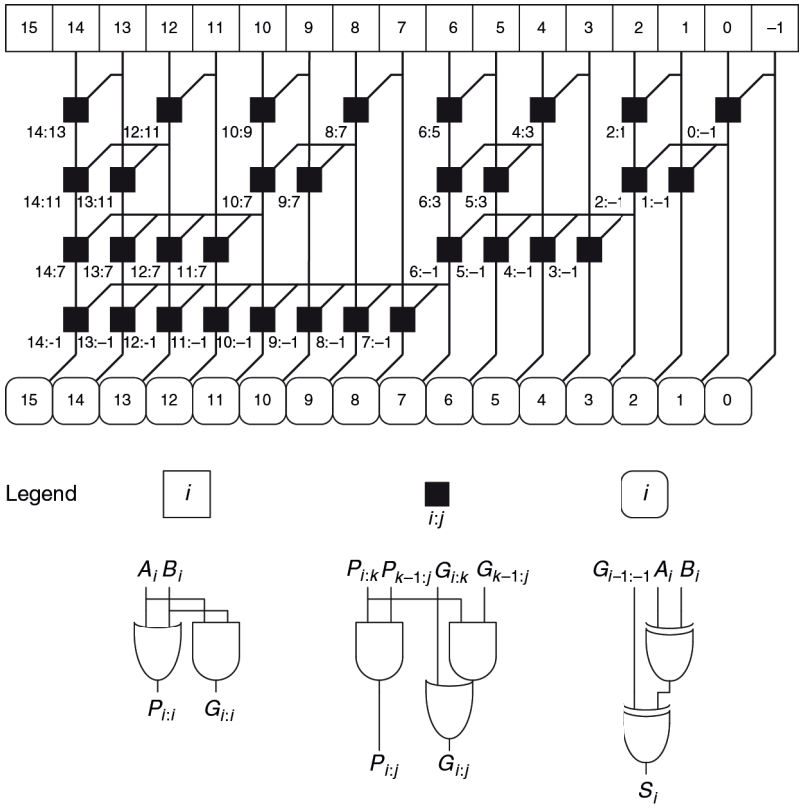


Рис. 5.7 16-разрядный префиксный сумматор

Поскольку время вычислений растет логарифмически с ростом количества входов, использование префиксного дерева является мощной технологией. При некотором умении этот принцип может быть применен для многих других схем (например, в **упражнении 5.7**).

Критический путь N-разрядного префиксного сумматора включает в себя предварительное вычисление P_i и G_i, за которым следует $\log_2 N$ каскадов черных ячеек для получения всех префиксов. Затем сигналы $G_{i\text{-}1:\text{-}1}$ обрабатываются финальными элементами «исключающее ИЛИ» в нижней части схемы для получения сигнала S_i. Задержка N-разрядного префиксного сумматора равна

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}, \tag{5.11}$$

где t_{pg_prefix} – задержка черной префиксной ячейки.

Пример 5.2 ЗАДЕРЖКА ПРЕФИКСНОГО СУММАТОРА

Рассчитайте задержку 32-разрядного префиксного сумматора при условии, что задержка каждого двухвходового логического элемента равна 100 пс.

Решение Задержка распространения каждой черной префиксной ячейки равна t_{pg_prefix} = 200 пс (задержки двух логических элементов). Таким образом, используя **уравнение (5.11)**, задержка распространения 32-разрядного префиксного сумматора равна 100 пс + $\log_2(32)$ + 200 пс + 100 пс + 1.2 нс, что примерно в 3 раза меньше, чем у сумматора с ускоренным переносом из **примера 5.1**. В действительности выгода не такая большая, но префиксные сумматоры на самом деле работают существенно быстрее, чем любые другие.

Краткие выводы к подразделу

В этом разделе были рассмотрены полусумматор, полный сумматор и три типа сумматоров с распространяющимся переносом: сумматоры с последовательным переносом, ускоренным переносом и префиксный сумматор. Быстрые сумматоры используют больше аппаратных ресурсов и, следовательно, являются более дорогостоящими и энергозатратными. Все это должно быть учтено при выборе нужного сумматора в процессе разработки.

Языки описания аппаратуры предоставляют возможность использования операции сложения для определения сумматора с распространяющимся переносом. Современные средства синтеза выбирают из множества возможных реализаций проекта самую дешевую и простую, которая удовлетворяет требованиям по скорости. Это очень упрощает работу разработчика. В **HDL-примере 5.1** с помощью языков описания аппаратуры реализован сумматор с распространяющимся переносом, имеющий вход и выход переноса.

HDL-пример 5.1 СУММАТОР

SystemVerilog

```
module adder #(parameter N = 8)
              (input logic [N-1:0]  a, b,
               input logic          cin,
               output logic [N-1:0] s,
               output logic         cout);

   assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
   generic(N: integer := 8);
   port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
        cin:  in  STD_LOGIC;
        s:    out STD_LOGIC_VECTOR(N-1 downto 0);
        cout: out STD_LOGIC);
   end;

   architecture synth of adder is
     signal result: STD_LOGIC_VECTOR(N downto 0);
   begin
     result <= ("0" & a) + ("0" & b) + cin;
     s      <= result(N-1 downto 0);
     cout   <= result(N);
   end;
```

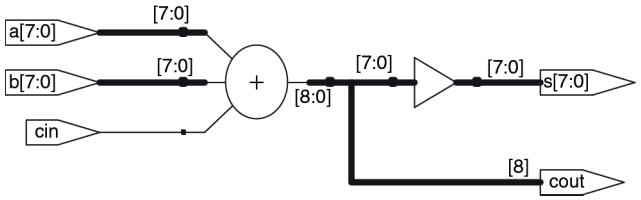


Рис. 5.8 **Синтезированный сумматор**

5.2.2. Вычитание

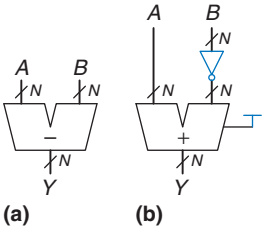


Рис. 5.9 **Устройство вычитания: (a) условное обозначение, (b) реализация**

В **разделе 1.4.6** было показано, что сумматоры могут складывать положительные и отрицательные числа, используя представление числа в дополнительном коде. Вычитание производится почти так же просто: меняется знак второго числа, затем числа складываются. Изменение знака числа в дополнительном коде производится путем инверсии битов и прибавления 1.

Для вычисления $Y = A - B$ вначале создается дополнительный код числа B: инвертируются разряды B и прибавляется 1; $-B = \overline{B} + 1$. Полученное значение складывается с A. Эта сумма может быть получена одним сумматором с распространяющимся переносом путем сложения $A + \overline{B}$ при $C_{in} = 1$. На **рис. 5.9** показано условное обозначение устройства вычитания и базовая аппаратная реализация для вычисления $Y = A - B$.

HDL-пример 5.2 описывает операцию вычитания.

HDL-пример 5.2 УСТРОЙСТВО ВЫЧИТАНИЯ

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic [N-1:0] y);

    assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity subtractor is
   generic(N: integer := 8);
   port(a, b:  in  STD_LOGIC_VECTOR(N-1 downto 0);
        y:     out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

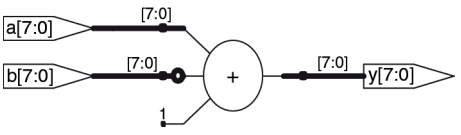


Рис. 5.10 **Синтезированное устройство вычитания**

5.2.3. Компараторы

Компараторы определяют, являются ли два двоичных числа равными или одно из них больше/меньше другого. Компаратор получает два N-разрядных двоичных числа A и B. Существует два типа компараторов:

- *компаратор равенства* выдает один выходной сигнал, показывая, равны ли A и B (A==B);
- *компаратор величины* выдает один и более выходных сигналов, показывая отношение величин A и B.

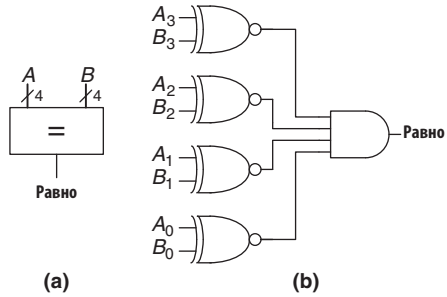


Рис. 5.11 4-разрядный компаратор равенства: (a) условное обозначение, (b) реализация

Компаратор равенства имеет простую аппаратную реализацию. На **рис. 5.11** показано обозначение и реализация 4-разрядного компаратора равенства. Сначала с помощью логических элементов XNOR он проверяет, являются ли соответствующие разряды A и B равными. Значения будут равными, если все соответствующие разряды равны.

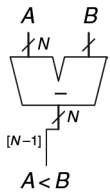


Рис. 5.12 *N*-разрядный компаратор для сравнения двух чисел с учетом знака

Как показано на **рис. 5.12**, компаратор величины вычисляет $A - B$ и анализирует знак (самый старший разряд) результата. Если результат отрицательный (самый старший разряд = 1), то A меньше B. В противном случае A больше или равно B.

HDL-пример 5.3 показывает использование этих двух типов компараторов.

HDL-пример 5.3 КОМПАРАТОРЫ

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic eq, neq, lt,
                           lte, gt, gte);
  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a <b);
  assign lte = (a <= b);
  assign gt  = (a >b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity comparators is
  generic(N: integer : = 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;
architecture synth of comparator is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

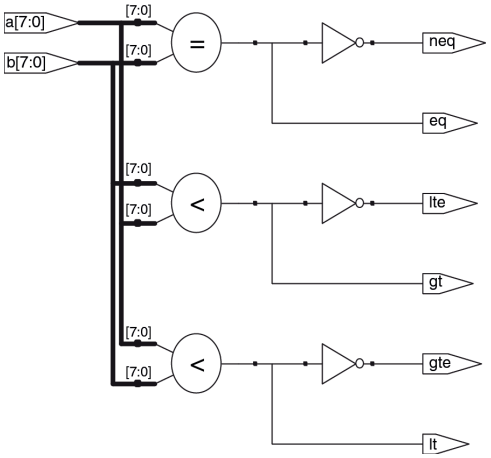


Рис. 5.13 **Синтезированный компаратор**

5.2.4. Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) (Arithmetic / Logical Unit, ALU) объединяет различные арифметические и логические операции в одном узле. Например, типичное АЛУ может выполнять сложение, вычитание, сравнение величин, операции И и ИЛИ. АЛУ входит в ядро большинства компьютерных систем.

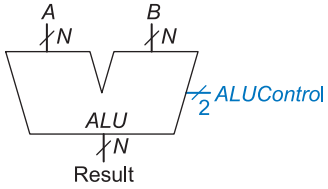


Рис. 5.14 **Условное обозначение АЛУ**

На **рис. 5.14** показано условное обозначение N-разрядного АЛУ с N-разрядными входами и выходами. В АЛУ поступает управляющий сигнал F, который определяет, какую функцию нужно выполнить. Обычно сигналы управления показывают голубым цветом, чтобы отличать их от сигналов данных. В **табл. 5.1** перечислены типичные функции, которые выполняет АЛУ.

Таблица 5.1 **Операции АЛУ**

$ALUControl_{1:0}$	Действие
00	Сложение
01	Вычитание
10	И
11	ИЛИ

На **рис. 5.15** показана реализация блока АЛУ. Он состоит из N-разрядного сумматора и N двухвходовых логических элементов И и ИЛИ. Также он содержит инверторы и мультиплексор для инверсии битов вхо-

да B, когда активен управляющий сигнал $ALUControl_0$. Мультиплексор с организацией 4:1 выбирает необходимую функцию, исходя из сигналов управления $ALUControl$.

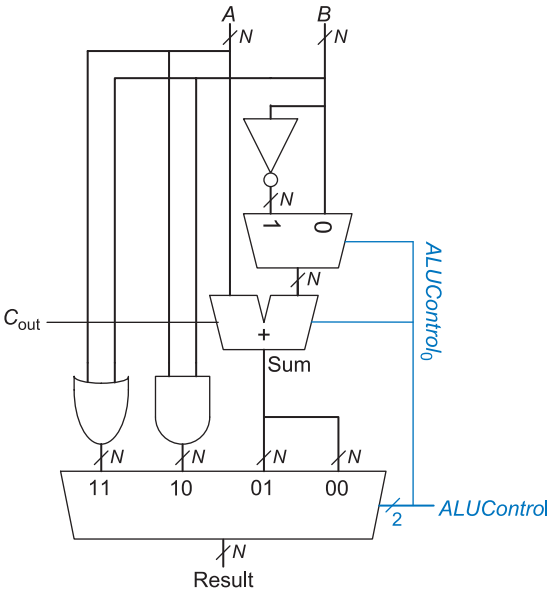


Рис. 5.15 *N*-разрядное АЛУ

Говоря точнее, если $ALUControl = 00$, выходной мультиплексор выбирает операцию $A + B$. Если $ALUControl = 01$, АЛУ выполняет операцию $A - B$. (В разделе 5.2.2 мы говорили о том, что при использовании дополнительного кода $\overline{B} + 1 = -B$. Поскольку $ALUControl_0 = 1$, на входы сумматора поступают значения A и $\overline{B}$ и активный сигнал переноса, заставляя его выполнять вычитание: $A + \overline{B} + 1 = A - B$.) Если $ALUControl = 10$, то АЛУ выполняет операцию A AND B. Если $ALUControl = 11$, АЛУ выполняет операцию A OR B.

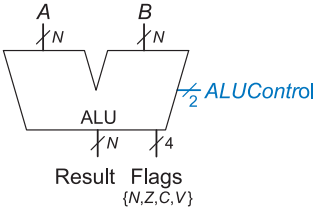


Рис. 5.16 Условное обозначение АЛУ с флагами результата

У некоторых АЛУ предусмотрены специальные выходные сигналы, называемые *флагами*, которые отражают состояние выхода *Result* АЛУ. На **рис. 5.16** показано условное обозначение АЛУ с 4-битным выходом сигналов флагов. Как показано на схеме этого АЛУ на **рис. 5.17**, выход *Flags* содержит флаги N, Z, C и V, которые указывают, соответственно, что после завершения операции АЛУ на выходе *Result* получилось отрицательное (N, negative) или нулевое (Z, zero) значение или что в сумматоре произошел перенос (C, carry) или переполнение (V, overflow). Напомним, что старший бит числа, представленного в дополнительном двоичном коде, равен 1, если число

отрицательное, и 0 в иных случаях. Следовательно, флаг N копирует состояние старшего бита $Result_{31}$ на выходе АЛУ. Флаг Z становится равен единице, когда все разряды результата равны нулю (это состояние детектируется N-разрядным логическим элементом ИЛИ-НЕ, как показано на **рис. 5.17**). Флаг C становится равен единице, когда сумматор выполняет перенос, а АЛУ выполняет сложение или вычитание ($ALUControl_1 = 0$).

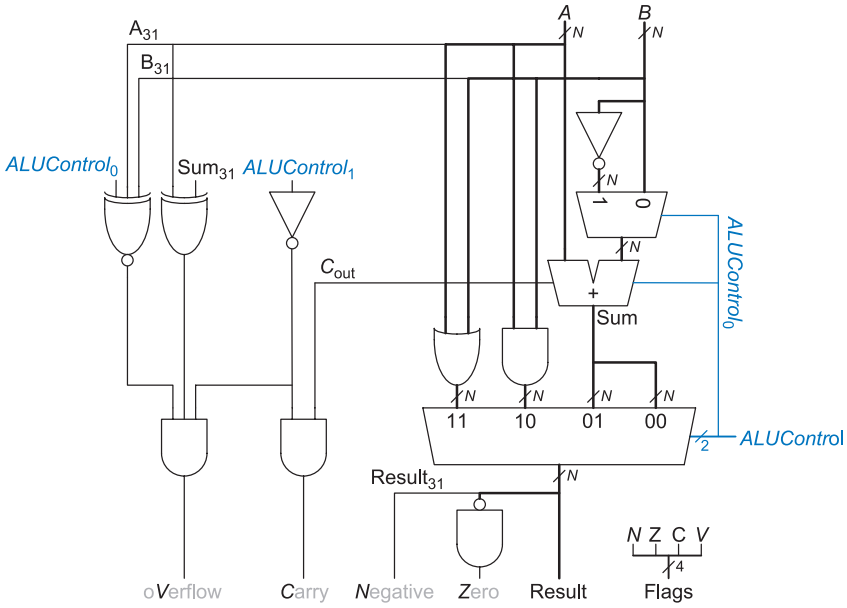


Рис. 5.17 Схема *N*-разрядного АЛУ с флагами результата

Схема обнаружения переполнения, показанная в левой части **рис. 5.17**, несколько сложнее. В **разделе 1.4.6** мы говорили, что переполнение происходит, когда сложение двух чисел с одинаковым знаком дает результат с противоположным знаком. Поэтому флаг V становится равен единице, когда выполняются все три следующих условия: (1) АЛУ выполняет сложение или вычитание ($ALUControl_1 = 0$), (2) A и Sum имеют противоположные знаки, что обнаружено логическим элементом XOR, и (3) возможно переполнение. То есть, исходя из логики работы логического элемента XNOR, либо A и B имеют одинаковый знак, и сумматор выполняет сложение ($ALUControl_0 = 0$), либо A и B имеют противоположные знаки, и сумматор выполняет вычитание ($ALUControl_0 = 1$). Логический элемент И с тремя входами определяет, когда все три условия верны, и устанавливает флаг V.

Флаги АЛУ также можно использовать для операции сравнения, как показано в **табл. 5.2**. Чтобы сравнить числа A и B, АЛУ выполняет вычитание $A - B$ и проверяет флаги. Если установлен флаг Z, то результат вычитания равен нулю и $A = B$. В противном случае $A \neq B$.

Таблица 5.2 **Сравнение с учетом знака и без учета знака**

Сравнение	С учетом знака	Без учета знака
$=$	Z	Z
$\neq$	$\overline{Z}$	$\overline{Z}$
$<$	$N \oplus V$	$\overline{C}$
$\leq$	$Z + (N \oplus V)$	$Z + \overline{C}$
$>$	$\overline{Z} \cdot (N \oplus V)$	$\overline{Z} \cdot C$
$\geq$	$(N \oplus V)$	C

Сравнение по величине (magnitude comparison) выглядит сложнее и зависит от того, с числами в каком формате выполняется операция (знаковыми или беззнаковыми). Например, чтобы определить истинность условия $A < B$, мы вычисляем $A - B$ и проверяем, является ли результат отрицательным. Если числа представлены в беззнаковом формате, результат сравнения будет отрицательным, если нет бита переноса[1]. Если числа представлены в формате со знаком, мы не можем полагаться на перенос, потому что маленькие отрицательные числа выглядят так же, как большие положительные числа без знака. Вместо этого мы просто вычисляем $A - B$ и проверяем, является ли ответ отрицательным, на что указывает флаг N. При этом в случае переполнения состояние флага N будет некорректным. Следовательно, A меньше B, если $N \oplus V$ (другими словами, если результат отрицательный и нет переполнения или если результат положительный, но произошло переполнение). Таким образом, мы можем сгенерировать сигнал L (less than, меньше чем), если $A < B$. Для чисел в беззнаковом формате $L = \overline{C}$. Для чисел со знаком $L = N \oplus V$. Остальные проверки не столь сложны. Результат сравнения «меньше или равно» ($\leq$) – это L OR Z, потому что L означает «меньше», а Z означает «равно». Результат сравнения «больше или равно» ($\geq$) является инверсией сигнала «меньше» ($\overline{L}$). Результат сравнения «больше» ($>$) получается следующим образом: $\overline{L}$ AND $\overline{Z}$.

Пример 5.3 ОПЕРАЦИЯ СРАВНЕНИЯ

Рассмотрим два значения $A = 1111$ и $B = 0010$. Определите, выполняется ли условие $A < B$, сначала рассматривая эти значения как числа без знака (15 и 2), а затем как числа со знаком (-1 и 2).

[1] Вы можете убедиться в этом, попробовав сравнить несколько чисел. В качестве альтернативного доказательства отметим тот факт, что инверсию знака (т. е. представление в дополнительном коде) N-битных чисел для вычитания можно записать как $-B = \overline{B} + 1 = 2^N - B$. Следовательно, $A + (-B) = 2^N + A - B$. Эта операция приведет к переносу (1 в столбце N), если $A \geq B$, и к отсутствию переноса, если $A < B$.

Решение Вычислим $A - B = A + \overline{B} + 1 = 1111 + 1101 + 1 = 11101$. Флаг переноса C равен единице, на что указывает разряд, выделенный синим цветом. Флаг N тоже равен единице, на что указывает разряд, выделенный курсивом. Флаг V равен нулю, потому что результат имеет тот же знаковый бит, что и A. Флаг Z равен нулю, потому что результат не равен 0000.

При сравнении чисел без знака $L = \overline{C} = 0$, потому что 15 не меньше 2. При сравнении со знаком $L = N \oplus V = 1$, потому что -1 меньше 2.

Некоторые АЛУ также поддерживают команду SLT (set if less than – установить, если меньше, чем). Когда $A < B$, $Result = 1$. В противном случае $Result = 0$. Эта команда полезна для компьютеров, у которых нет доступа к флагам АЛУ, поскольку фактически позволяет передать информацию о флагах в $Result$. При выполнении команды SLT обычно АЛУ рассматривает входные данные как числа со знаком.

В беззнаковом варианте этой команды (SLTU) АЛУ рассматривает входные данные как числа без знака. Существует множество вариантов базового АЛУ, выполняющего и другие операции, такие как NOT, XOR или XNOR. Разработку HDL-кода для N-разрядного АЛУ, включая версии, поддерживающие команду SLT и флаги результата, мы оставили для **упражнений 5.11–5.14**.

Пример 5.4 МОДИФИКАЦИЯ АЛУ ДЛЯ ВЫПОЛНЕНИЯ КОМАНДЫ SLT

Модифицируйте АЛУ для выполнения команды SLT.

Решение Чтобы реализовать в АЛУ поддержку еще одной операции, необходимо добавить в мультиплексор пятый канал. Мы выполняем проверку $A < B$ при помощи операции $A - B$: если результат отрицательный, то действительно A меньше B. В **табл. 5.3** представлен дополненный одним разрядом сигнал $ALUControl$ для обработки команды SLT, а на **рис. 5.18 (a)** показана модифицированная схема, в которой изменения выделены синим и черным цветами. Для операции SLT мы используем управляющий сигнал $ALUControl = 101$, а разряд $ALUControl_0 = 1$ заставляет сумматор выполнять операцию вычитания $A - B$. Если сигнал $Sum_{N-1} = 1$, это означает, что результат операции $A - B$ отрицательный, т. е. $A < B$. Затем мы расширяем значение Sum_{N-1} нулями и подаем его на вход мультиплексора 101 для завершения операции SLT. Обратите внимание, что эта схема не учитывает переполнение. Когда происходит переполнение, у числа Sum будет неправильный знак. Поэтому мы выполняем операцию XOR между знаковым битом Sum и флагом переполнения V, чтобы правильно показать отрицательное значение Sum, как изображено на **рис. 5.18 (b)**.

Таблица 5.3 Управляющие сигналы АЛУ с поддержкой SLT

$ALUControl_{2:0}$	Операция
000	Сложение
001	Вычитание
010	AND
011	OR
101	SLT

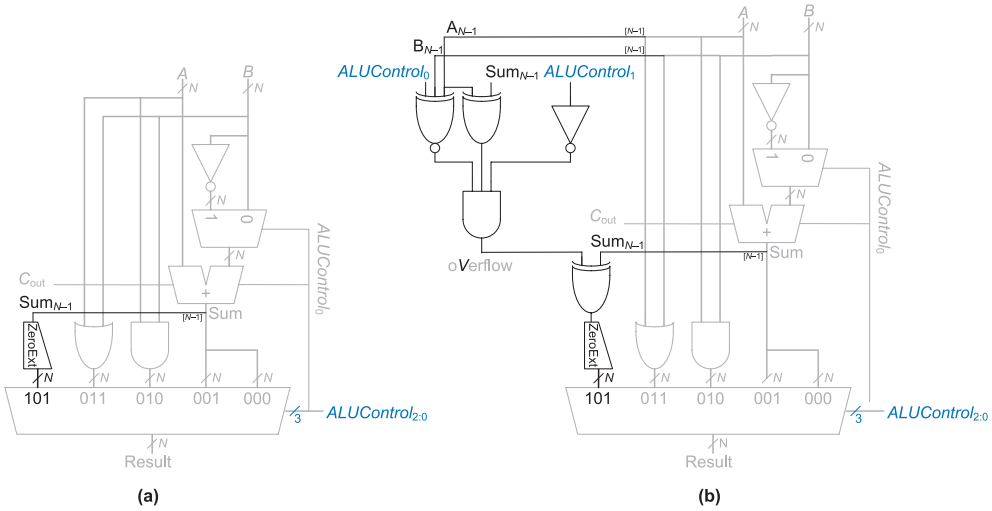


Рис. 5.18 **Схема АЛУ с поддержкой SLT (a) без учета переполнения, (b) с учетом переполнения**

5.2.5. Схемы сдвига и циклического сдвига

Схемы *сдвига и схемы циклического сдвига* перемещают биты и, следовательно, умножают или делят число на степень 2. В соответствии с названием схемы сдвига передвигают разряды двоичного числа влево или вправо на определенное число позиций. Существует несколько видов таких схем.

- **Логические схемы сдвига** сдвигают число влево (LSL) или вправо (LSR) и заполняют пустые разряды нулями.

  Например, 11001 LSR 2 = 00110; 11001 LSL 2 = 00100.

- **Арифметические схемы сдвига** действуют так же, как и логические, но при сдвиге вправо они заполняют наиболее значащие разряды значением знакового бита исходного числа. Это необходимо при умножении и делении чисел со знаком (**разделы 5.2.6** и **5.2.7**). Арифметический сдвиг влево (ASL) работает так же, как и логический (LSL).

  Например: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100.

- **Схемы циклического сдвига** сдвигают число по кругу так, что пустые места заполняются разрядами, которые выдвинуты на другом конце числа.

  Например: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111.

N-разрядная схема сдвига может быть построена из N мультиплексоров N:1. Вход сдвигается на 0–(N – 1) разрядов в зависимости от значения log_2N линий выбора. На **рис. 5.19** показаны условное обозначение и аппаратная реализация 4-разрядной схемы сдвига. Операторы <<, >> и >>> обычно обозначают сдвиг влево, логический сдвиг вправо и арифметический сдвиг вправо соответственно. В зависимости от значения 2-разрядной величины сдвига $shamt_{1:0}$ на выход Y поступает входной сигнал A, сдвинутый на 0–3 разряда. Для всех схем сдвига если $shamt_{1:0} = 00$, то $Y = A$. В **упражнении 5.14** рассматривается разработка схем циклического сдвига.

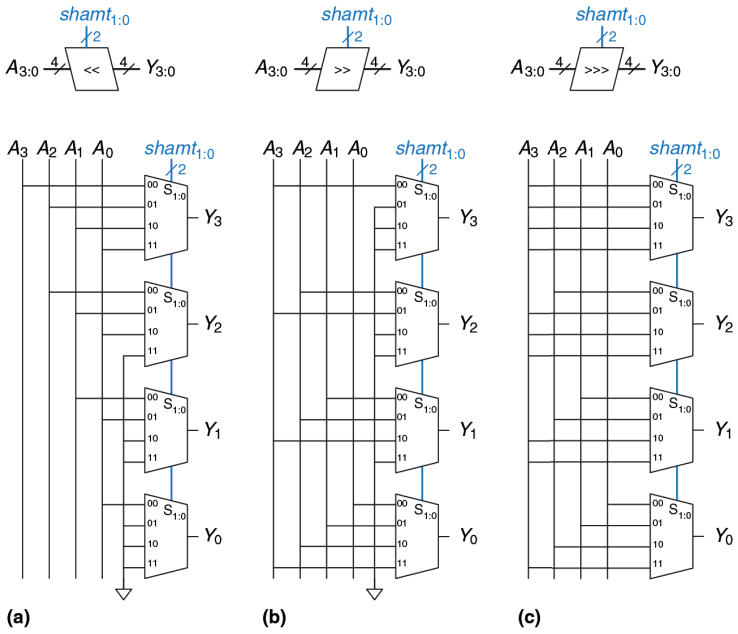


Рис. 5.19 4-разрядные схемы сдвига: (a) сдвиг влево, (b) логический сдвиг вправо, (c) арифметический сдвиг вправо

Сдвиг влево – это частный случай умножения. Сдвиг влево на N бит умножает число на 2^N. Например, 000011_2 << 4 = 110000_2 равносильно $3_{10} \times 2^4 = 48_{10}$.

Арифметический сдвиг вправо – это специальный случай деления. Арифметический сдвиг вправо на N бит делит число на 2^N. К примеру, 11100_2 >>> 2 = 11111_2 равносильно $-4_{10}/2^2 = -1_{10}$.

5.2.6. Умножение

Умножение беззнаковых двоичных чисел аналогично десятичному умножению, но оперирует только с единицами и нулями. На **рис. 5.20** сравнивается умножение двоичных и десятичных чисел. В обоих случаях

частичные произведения формируются путем умножения отдельных разрядов множителя на все множимое. Сдвинутые частичные произведения затем складываются, и мы получаем результат.

```
   230      множимое          0101
 ×  42      множитель       × 0111
 ----                       ------
  460      частные            0101
+ 920      произведения       0101
 ----                         0101
 9660                       + 0000
                            ------
          результат         0100011
```

2 30 × 42 = 9660 (a) 5 × 7 = 35 (b)

Рис. 5.20 Умножение: (a) десятичное, (b) двоичное

В общем случае умножитель $N \times N$ перемножает два N-разрядных числа и порождает $2N$-разрядный результат. Частичные произведения при двоичном умножении равны или множимому, или нулю. Умножение одного разряда двоичных чисел равносильно операции И, поэтому для формирования частичных произведений используются логические элементы И.

На **рис. 5.21** показаны условное обозначение, функциональное описание и схема умножителя 4×4. Умножитель получает множимое и множитель A и B и вычисляет произведение P. На **рис. 5.21 (b)** показано, как формируются частичные произведения. Каждое частичное произведение равно результату операций И, аргументами которых являются отдельные разряды множителя (B_3, B_2, B_1 или B_0) и все разряды множимого (A_3, A_2, A_1, A_0). Для N-разрядных операндов будет существовать N частичных произведений и $N - 1$ каскадов (стадий) одноразрядных сумматоров. Например, для умножителя 4×4 частичное произведение первого ряда – это B_0 AND (A_3, A_2, A_1, A_0). Это частичное произведение прибавляется к сдвинутому второму частичному произведению B_1 AND (A_3, A_2, A_1, A_0). Следующие ряды логических элементов И и сумматоров формируют и добавляют оставшиеся частичные произведения.

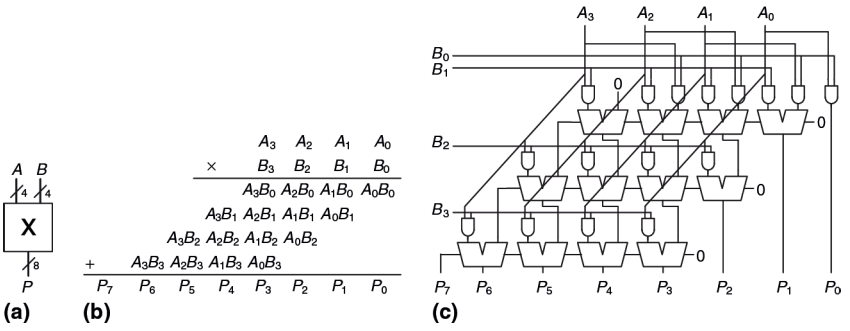


Рис. 5.21 Умножитель 4×4: (a) условное обозначение, (b) функциональное описание, (c) схема

Код HDL для знаковых и беззнаковых умножителей показан в **HDL-примере 4.33**. Как и в случае с сумматорами, существует множество схем умножителей с разным соотношением быстродействие/стоимость. Инструменты синтеза могут выбрать наиболее подходящую схему с учетом требований к быстродействию.

При выполнении операции *умножения с накоплением* АЛУ перемножает два числа и прибавляет результат к третьему числу – обычно к накопленному значению. Такие операции, также называемые MAC (multiply accumulate), часто используются в алгоритмах *цифровой обработки сигналов* (digital signal processing, DSP), таких как преобразование Фурье, которое требует суммирования произведений.

5.2.7. Деление

Двоичное деление N-разрядных беззнаковых чисел в диапазоне $[0, 2^{N-1}]$ может быть выполнено с использованием следующего алгоритма:

```
R' = 0
for i = N-1 to 0
   R = {R' << 1, Ai}
   D = R - B
   if D < 0 then  Qi = 0, R' = R  // R < B
   else           Qi = 1, R' = D  // R ≥ B
R = R'
```

Частичный остаток R инициализируется 0. Наиболее значимый разряд делимого A затем становится наименее значимым разрядом R. Делитель B многократно вычитается из частичного остатка, и определяется знак разницы D. Если она отрицательная (т. е. знаковый разряд равен 1), то разряд частного Q_i равен 0, и разница отбрасывается. В противном случае Q_i равен 1 и частичный остаток обновляется, он становится равным разнице D. Затем частичный остаток удваивается (сдвигается влево на один разряд), и процесс повторяется. Результат удовлетворяет условию $A/B = Q + R/B$.

На **рис. 5.20** показана схема 4-разрядной матрицы деления.

Схема вычисляет A/B и на выход выдает частное Q и остаток R. На вставке показаны условные обозначения и схемы каждого блока в матрице деления. Сигнал N показывает, является ли результат $R - B$ отрицательным. Это определяется по выходному сигналу D самого левого блока в ряду, который является знаком разницы.

Задержка N-разрядной матрицы деления увеличивается пропорционально N^2, так как перенос должен пройти через все N каскадов в ряду, перед тем как определится знак и мультиплексор выберет R или D. Это повторяется для всех N рядов. Деление – очень медленная и дорогая операция в аппаратной реализации, поэтому ее следует использовать как можно реже.

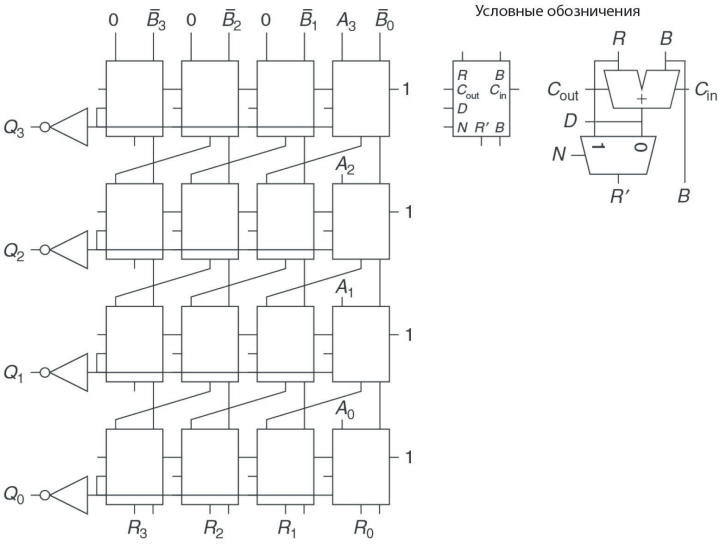


Рис. 5.22 **Матрица деления**

5.2.8. Дополнительная литература

Компьютерная арифметика может быть предметом целой книги. В учебнике *Digital Arithmetic* (M. Ercegovac, T. Lang; 2003) представлен отличный обзор по данной теме. Учебник *CMOS VLSI Design* (A. Weste, D. Harris; 2010) охватывает проектирование высокопроизводительных схем для арифметических операций.

5.3. Представление чисел

Компьютер работает как с целыми, так и с дробными числами. До настоящего момента мы рассматривали только представления знаковых и беззнаковых целых чисел, которые были описаны в **разделе 1.4**. В данном разделе вводится представление чисел с фиксированной и с плавающей запятой, с помощью которого можно представить рациональные числа. Числа с фиксированной запятой – это аналог десятичных чисел; некоторые биты представляют целую часть, а оставшиеся – дробную. Числа с плавающей запятой являются аналогом экспоненциального представления числа с мантиссой и порядком[1].

[1] В англоязычных странах в качестве разделителя целой и дробной частей чисел используется точка, а не запятая. В современной русскоязычной литературе могут встречаться оба термина. – *Прим. перев.*

5.3.1. Числа с фиксированной запятой

Представление «с фиксированной запятой» подразумевает двоичную запятую между битами целой и дробной частей, аналогично десятичной запятой между целой и дробной частями обычного десятичного числа.

Например, на **рис. 5.23 (a)** показано число с фиксированной запятой с четырьмя битами целой части и четырьмя дробной.

На **рис. 5.23 (b)** голубым цветом показана двоичная запятая, а на **рис. 5.23 (c)** изображено эквивалентное десятичное число.

Знаковые числа с фиксированной запятой могут использовать как прямой, так и дополнительный код.

На **рис. 5.24** показаны оба представления числа −2,375 с фиксированной запятой с использованием четырех целых бит и четырех дробных бит. Неявная двоичная запятая для ясности изображена голубым цветом.

В прямом коде знаковый бит используется для указания знака. Дополнительный код двоичного числа получается инверсией битов абсолютного значения и добавлением 1 к младшему разряду. В этом примере младший разряд соответствует 2^{-4}.

(a) 01101100

(b) 0110,1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6{,}75$

Рис. 5.23 Представление числа 6,75 с фиксированной запятой с четырьмя битами целой части и четырьмя — дробной

(a) 0010,0110

(b) 1010,0110

(c) 1101,1010

Рис. 5.24 Представление числа −2,375 с фиксированной запятой: (a) абсолютное значение, (b) прямой код, (c) дополнительный код

Как и все представления двоичных чисел, числа с фиксированной запятой являются лишь набором битов. Не существует способа узнать о существовании двоичной запятой, кроме как из соглашения между людьми, интерпретирующими число.

Пример 5.5 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ С ЧИСЛАМИ С ФИКСИРОВАННОЙ ЗАПЯТОЙ

Для корректных вычислений с использованием чисел с фиксированной запятой используется двоичное представление в дополнительном коде.

Вычислим выражение 0,75 + −0,625, используя числа с фиксированной запятой.

Решение Сначала преобразуем 0,625, абсолютное значение второго числа, в стандартное представление двоичного числа с фиксированной запятой. $0{,}625 \geq 2^{-1}$, следовательно, ставим 1 в разряд 2^{-1}, оставляя $0{,}625 - 0{,}5 = 0{,}125$. Так как $0{,}125 < 2^{-2}$, то ставим 0 в разряд 2^{-2}. Так как $0{,}125 \geq 2^{-3}$, то ставим 1 в разряд 2^{-3}, оставляя $0{,}125 - 0{,}125 = 0$. Таким образом, в разряде 2^{-4} будет 0. Таким образом, $0{,}625_{10} = 0000{,}1010_2$.

На **рис. 5.25** показано преобразование числа −0,625 в двоичное представление в дополнительном коде. На **рис. 5.26** показано сложение чисел с фиксированной запятой и, для сравнения, десятичный эквивалент.

Заметьте, что первый единичный бит в двоичном представлении числа с фиксированной запятой на **рис. 5.26 (a)** отброшен в 8-битовом результате.

```
  0000,1010   Двоичный модуль
  1111,0101   Дополнение до единицы
+        1    Прибавить 1
  1111,0110   Дополнение до двух
```

Рис. 5.25 Представление числа в дополнительном коде

```
  0000,1100          0,75
+ 1111,0110      +  (–0,625)
 10000,0010          0,125
     (a)              (b)
```

Рис. 5.26 Сложение: (a) двоичных чисел с фиксированной запятой, (b) десятичный эквивалент

5.3.2. Числа с плавающей запятой

Числа с плавающей запятой соответствуют экспоненциальному представлению. В этом представлении преодолены ограничения наличия только фиксированного количества целых и дробных битов, поэтому оно позволяет представлять очень большие и очень маленькие числа. Как и в экспоненциальном представлении, числа с плавающей запятой имеют знак, мантиссу (M), основание (B) и порядок (E), что показано на **рис. 5.27**.

$$\pm M \times B^E$$

Рис. 5.27 Числа с плавающей запятой

К примеру, число $4{,}1 \times 10^3$ является десятичным экспоненциальным представлением числа 4100. Мантиссой является 4,1, основание равно 10, а порядок равен 3. Десятичная запятая «переплывает» на позицию правее самого значимого (старшего) разряда. У чисел с плавающей запятой основание будет равно 2, а мантисса будет двоичным числом. 32 бита используются для представления 1 знакового бита, 8 бит порядка и 23 бит мантиссы.

Пример 5.6 32-БИТНОЕ ЧИСЛО С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Найдите представление десятичного числа 228 в виде числа с плавающей запятой.

Решение Для начала преобразуем десятичное число в двоичное: $228_{10} = 11100100_2 = 1{,}11001_2 \times 2^7$. На **рис. 5.28** показано 32-битное кодирование, которое далее для эффективности будет модифицировано. Знаковый бит положительный, равен 0, 8 бит порядка дают значение 7, а оставшиеся 23 бита – это мантисса.

Системы счисления с фиксированной запятой обычно используются в приложениях цифровой обработки сигналов (DSP), графики и машинного обучения, поскольку вычисления выполняются быстрее и на них расходуется меньше энергии, чем в системах с плавающей запятой. Q1.15 (также известный как Q15) – наиболее распространенный формат, в котором числа со знаком в диапазоне (−1, 1) представлены с точностью до 15 разрядов.

Формат Q1.31 (или просто Q31) иногда используется для хранения промежуточных результатов повышенной точности, например при вычислении быстрого преобразования Фурье. В формате U8.8 иногда передают показания датчиков, считываемых аналого-цифровыми преобразователями (АЦП). Обратите внимание, что все эти форматы упаковываются в 16- или 32-битные слова для эффективного хранения в ячейках компьютерной памяти, ширина которых обычно равна степени двойки.

Прим. от научного редактора: есть и другие способы представления чисел, такие как, например, система остаточных классов или формат posit.

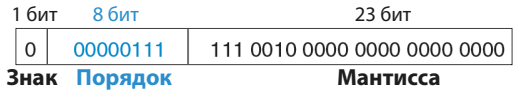


Рис. 5.28 32-разрядное кодирование числа с плавающей запятой: версия 1

В двоичных числах с плавающей запятой первый бит мантиссы (слева от запятой) всегда равен 1, и поэтому его можно не сохранять. Это называется *неявная старшая единица*. На **рис. 5.29** изображено модифицированное представление: $228_{10} = 11100100_2 \times 2^0 = 1{,}11001_2 \times 2^7$. Неявная старшая единица не входит в 23 бита мантиссы. Сохраняются только дробные биты. Это освобождает дополнительный бит для полезных данных.

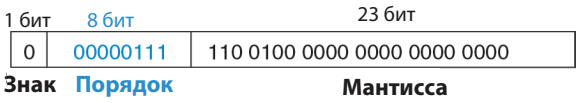


Рис. 5.29 Кодирование числа с плавающей запятой: версия 2

Очевидно, что существует много разумных способов представления чисел с плавающей запятой. Много лет производители компьютеров использовали несовместимые форматы. Результат от одного компьютера не мог быть непосредственно интерпретирован другим. Институт инженеров электротехники и электроники (Institute of Electricaland Electronics Engineers, IEEE) решил эту проблему, определив в 1985 году стандарт IEEE 754. Сейчас этот формат используется повсеместно. Именно он будет рассматриваться будет обсуждаться в данном разделе.

Сделаем последнюю модификацию представления порядка. Порядок должен представлять как положительный показатель степени, так и отрицательный. Для этого в формате с плавающей запятой используется смещенный порядок, который представляет собой первоначальный порядок плюс постоянное смещение. 32-битное представление с плавающей запятой использует смещение 127. Например, для порядка 7 смещенный порядок будет выглядеть так: $7 + 127 = 134 = 10000110_2$, для порядка −4 смещенный порядок равен $-4 + 127 = 123 = 01111011_2$.

На **рис. 5.30** показано представление числа $1{,}11001_2 \times 2^7$ в формате с плавающей запятой с неявной старшей единицей и смещенным порядком $134(7 + 127)$. Это представление соответствует стандарту IEEE 754.

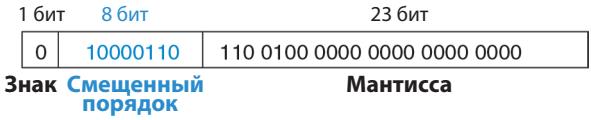


Рис. 5.30 Представление числа с плавающей запятой по стандарту IEEE 754

Особые случаи: 0, ±∞ и NaN

Стандарт IEEE для чисел с плавающей запятой включает в себя особые случаи представления таких чисел, как 0, бесконечность и недопустимое значение. К примеру, представить число 0 в виде числа с плавающей запя-

той невозможно из-за наличия неявной старшей единицы. Для этих случаев зарезервированы специальные коды: в таких кодах порядок состоит только из нулей или единиц. В **табл. 5.4** показано обозначение 0, ±∞ и NaN. Как и в знаковых числах, числа с плавающей запятой могут представлять как положительный, так и отрицательный 0. NaN используется для чисел, которые не существуют, например корень из −1 и $\log_2(-5)$.

Таблица 5.4 Обозначение 0, ±∞ и NaN в соответствии со стандартом IEEE 754

Число	Знак	Порядок	Мантисса
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Не ноль

Форматы с одинарной и двойной точностью

Ранее мы рассматривали 32-битные числа с плавающей запятой. Такой формат еще называют форматом с одинарной точностью. Стандарт IEEE 754 также определяет 64-битные числа с двойной точностью, которые позволяют представить больший диапазон чисел с большей точностью. В **табл. 5.5** приведено количество битов, используемых в полях разных форматов.

Таблица 5.5 Числа с плавающей запятой с одинарной и двойной точностью

Формат	Всего бит	Бит знака	Биты порядка	Биты мантиссы
одинарный	32	1	8	23
двойной	64	1	11	52

Если исключить специальные случаи, упомянутые ранее, обычные числа с одинарной точностью охватывают диапазон от $\pm 1{,}175494 \times 10^{-38}$ до $\pm 3{,}402824 \times 10^{38}$. Их точность составляет около 7 десятичных разрядов, так как $2^{-24} \approx 10^{-7}$. Числа с двойной точностью охватывают диапазон от $\pm 2{,}22507385850720 \times 10^{-308}$ до $\pm 1{,}79769313486232 \times 10^{308}$ и имеют точность около 15 десятичных разрядов.

Округление

Арифметические результаты, которые выходят за пределы доступной точности, необходимо округлять до наиболее близких чисел. Существуют следующие способы округления: округление в меньшую сторону (1), округление в большую сторону (2), округление до нуля (3) и округление к ближайшему числу (4). По умолчанию принято округление к ближайшему числу. В этом случае если два числа находятся на одинаковом рас-

Некоторые числа нельзя точно представить в виде числа с плавающей запятой, как, например, 1,7. Но когда вы вводите 1,7 на калькуляторе, вы видите точно 1,7, не 1,69999... Для этого большинство приложений, как, например, калькулятор и различные финансовые программы, используют двоично-десятичный формат (BCD), или формат с основанием 10. Числа в таком формате кодируют каждый десятичный разряд с помощью 4 бит со значением от 0 до 9. Например, число 1,7 в формате BCD с четырьмя целыми и четырьмя дробными битами представляет собой 0001.0111. Конечно, не все так просто. Ценой является усложнение арифметических схем и неполное использование кодировки (не используются кодировки A–F), следовательно, снижается эффективность. Таким образом, для ресурсоемких приложений числа в формате с плавающей запятой гораздо эффективнее.

стоянии, то выбирается то, у которого будет ноль в младшем разряде дробной части.

Напомним, что число *переполняется*, когда его величина слишком велика для какого-либо представления. Аналогично число является *изчезающе малым*, когда оно слишком мало для представления. При округлении (4) переполненные числа округляются до $\pm\infty$, а исчезающе малые округляются до нуля.

Сложение чисел с плавающей запятой

Сложение чисел с плавающей запятой – не такая простая операция, как в случае представления чисел в дополнительном коде. Для выполнения сложения двух таких чисел необходимо выполнить следующие шаги:

1. Выделить биты порядка и мантиссы.
2. Присоединить неявную старшую единицу к мантиссе.
3. Сравнить порядки.
4. При необходимости сдвинуть мантиссу числа, имеющего меньший порядок.
5. Сложить мантиссы.
6. При необходимости нормализовать мантиссу и порядок.
7. Округлить результат.
8. Собрать обратно порядок и мантиссу в итоговое число с плавающей запятой.

Вычисления при использовании чисел в формате с плавающей запятой обычно выполняются с помощью специальных аппаратных средств для увеличения скорости. Такая аппаратура называется FPU (floating-point unit). Она, как правило, отличается от CPU (central processing unit). Печально известный баг FDIV (floating-point division) в FPU процессора Pentium стоил компании Intel $475 млн, которые она вынуждена была потратить на отзыв и замену дефектных микросхем. Ошибка произошла только потому, что была неправильно загружена таблица преобразования.

На **рис. 5.31** показан процесс сложения чисел с плавающей запятой 7,875 ($1{,}11111\times2^{2}$) и 0,1875 ($1{,}1\times2^{-3}$). Результат равен 8,0625 ($1{,}0000001\times2^{3}$). После извлечения мантиссы и порядка, присоединения неявной старшей единицы (шаги 1 и 2) порядки сравниваются путем вычитания меньшего порядка из большего. Результатом будет число битов, на которое необходимо сдвинуть мантиссу меньшего числа вправо (шаг 4) для выравнивания двоичной запятой (т. е. чтобы сделать порядки равными). Выровненные значения складываются. Так как мантисса суммы больше или равна 2,0, результат нужно нормализовать, сдвинув его вправо на 1 бит и увеличив порядок на 1. В этом примере результат точный и никаких округлений не требуется. Результат сохраняется в формате с плавающей запятой после удаления неявной старшей единицы мантиссы и добавления знакового бита.

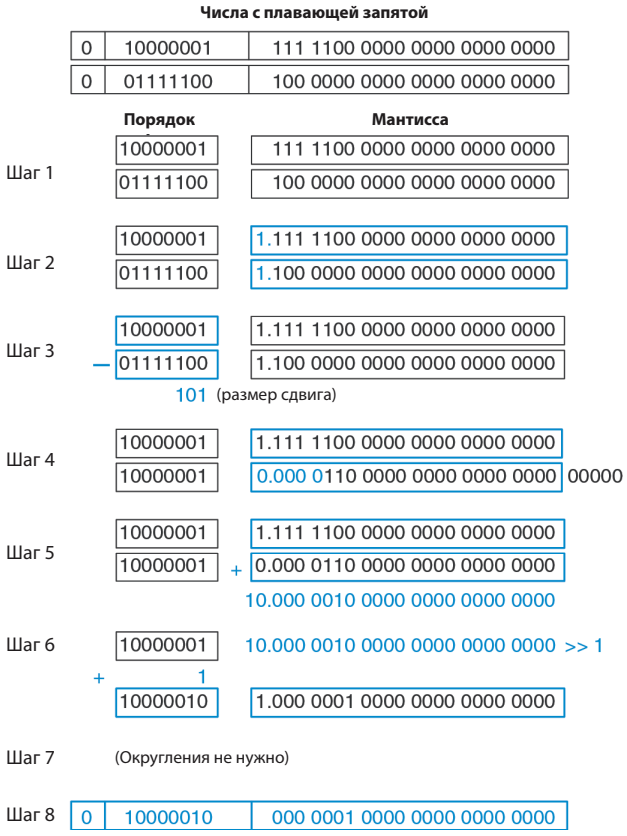


Рис. 5.31 Сложение чисел с плавающей запятой

5.4. Функциональные узлы последовательностной логики

В этом разделе будут рассмотрены функциональные узлы последовательной логики – счетчики и сдвиговые регистры.

5.4.1. Счетчики

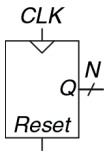


Рис. 5.32 Условное обозначение счетчика

N-разрядный двоичный счетчик, который показан на **рис. 5.32**, представляет собой последовательностную арифметическую схему, у которой есть входы тактового сигнала, сброса и N-разрядный выход Q. Сигнал сброса (*Reset*) инициализирует выходы нулевым значением. Выход счетчика по очереди принимает все 2^N возможных значений N-разрядного двоичного числа, переход к следующему значению происходит по переднему фронту тактового импульса.

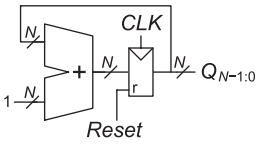


Рис. 5.33 *N*-разрядный счетчик

На **рис. 5.33** показан N-битный счетчик, состоящий из сумматора и регистра со сбросом. В каждом цикле счетчик добавляет 1 к значению, хранящемуся в регистре. В **HDL-примере 5.4** описан двоичный счетчик с асинхронным сбросом, а на **рис. 5.34** показана его синтезированная схема.

Старший разряд N-разрядного счетчика меняет свое значение через каждые 2^N тактов. Следовательно, такой счетчик снижает частоту тактовых импульсов в 2^N раз. Поэтому он называется *счетчиком-делителем на* 2^N и применяется для снижения частоты импульсов. Например, если цифровая схема имеет внутренний источник тактовых импульсов с частотой 50 МГц, при помощи 24-разрядного счетчика можно получить импульсы с частотой следования $(50{\times}10^6\ \text{Гц} / 2^{24}) = 2{,}98$ Гц. Человеческий глаз легко замечает мигание светодиода с такой частотой.

Еще одним популярным применением счетчика для формирования произвольных частот является *генератор с цифровым управлением* (digitally controlled oscillator, DCO, **пример 5.7**). Возьмем N-разрядный счетчик, значение которого с каждым тактом вместо единицы увеличивается на некое число p. Если на счетчик поступают тактовые импульсы с частотой f_{clk}, то старший разряд теперь меняет состояние с частотой $f_{out} = f_{clk} \times p / 2^N$. Путем подбора параметров p и N вы можете получить выходной сигнал любой частоты. Чем больше N, тем ближе фактическая частота генератора к искомому значению за счет больших аппаратных затрат.

HDL-пример 5.4 СЧЕТЧИК

SystemVerilog

```
module counter #(parameter N = 8)
                (input logic clk,
                 input logic reset,
                 output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)q <= 0;
    else      q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity counter is
  generic(N: integer := 8);
  port(clk, reset: in STD_LOGIC;
       q:          out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process(clk, reset) begin
    if reset then               q <= (OTHERS => '0');
    elsif rising_edge(clk) then q <= q + '1';
    end if;
  end process;
end;
```

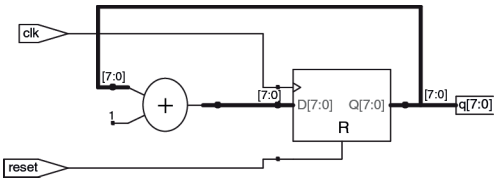


Рис. 5.34 **Синтезированная схема счетчика**

Пример 5.7 ГЕНЕРАТОР С ЦИФРОВЫМ УПРАВЛЕНИЕМ

Предположим, у вас есть источник опорных тактовых импульсов с частотой 50 МГц и вы хотите сгенерировать сигнал с частотой 500 Гц. Можно ли использовать для этой цели 24- или 32-разрядный счетчик? Какое значение p вам следует выбрать, и какие значения частоты, наиболее близкие к искомому значению 500 Гц, вы можете получить?

Решение Нам необходимо получить генератор с коэффициентом деления $p / 2^N = 500$ Гц / 50 МГц = 0,00001. Если $N = 24$, возьмем $p = 168$ и получим $f_{out} = 500{,}68$ Гц. Если $N = 32$, возьмем $p = 42950$ и получим $f_{out} = 500{,}038$ Гц.

5.4.2. Сдвиговые регистры

На **рис. 5.35** показан сдвиговый регистр (регистр сдвига, сдвигающий регистр), который имеет вход тактового сигнала, последовательный вход S_{in}, последовательный выход S_{out} и N параллельных выходов $Q_{N-1:0}$. По каждому переднему фронту тактового импульса в первый триггер регистра записывается новый бит со входа S_{in}, а содержимое следующих триггеров сдвигается вперед. Последний бит регистра можно считать с выхода S_{out}.

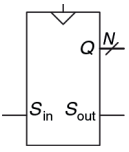


Рис. 5.35 **Условное обозначение сдвигового регистра**

Сдвиговый регистр можно рассматривать как последовательно-параллельный преобразователь. На вход S_{in} поступают последовательные данные (по одному биту за раз). После N циклов последние N значений входного сигнала можно параллельно считать с выхода Q.

Как показано на **рис. 5.36**, сдвиговый регистр может быть построен из N последовательно соединенных триггеров. Некоторые сдвиговые регистры имеют сигнал сброса для инициализации всех триггеров.

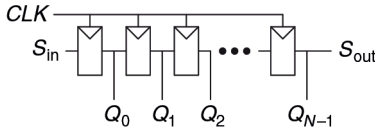


Рис. 5.36 **Схема сдвигающего регистра**

В *параллельно-последовательный преобразователь* параллельно загружается N бит, которые затем последовательно (по одному биту за раз) поступают на выход. Схемотехника параллельно-последовательного преобразователя и сдвигового регистра подобны. Сдвиговый регистр можно модифицировать для выполнения как последовательно-парал-

лельного, так и параллельно-последовательного преобразования, если к нему добавить параллельный вход $D_{N-1:0}$ и сигнал управления *Load*, как показано на **рис. 5.37**. Когда вход *Load* активирован, во все триггеры параллельно загружаются данные со входа *D*. В противном случае сдвиговый регистр выполняет обычный сдвиг. В **HDL-примере 5.5** сдвиговый регистр описан на языках HDL.

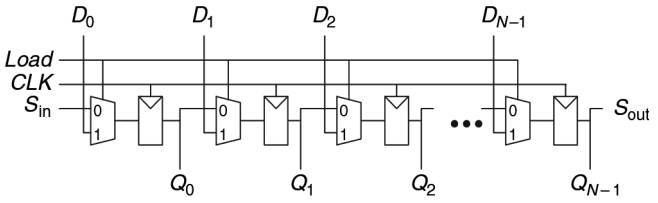


Рис. 5.37 Сдвиговый регистр с параллельной загрузкой

HDL-пример 5.5 СДВИГОВЫЙ РЕГИСТР С ПАРАЛЛЕЛЬНОЙ ЗАГРУЗКОЙ

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic         clk,
                  input  logic         reset, load,
                  input  logic         sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic         sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};
  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(N-1 downto 0);
       q:          out STD_LOGIC_VECTOR(N-1 downto 0);
       sout:       out STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process(clk, reset) begin
    if reset = '1' then  q <= (OTHERS => '0');
    elsif rising_edge(clk) then
      if load then       q <= d;
      else               q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;
  sout <= q(N-1);
end;
```

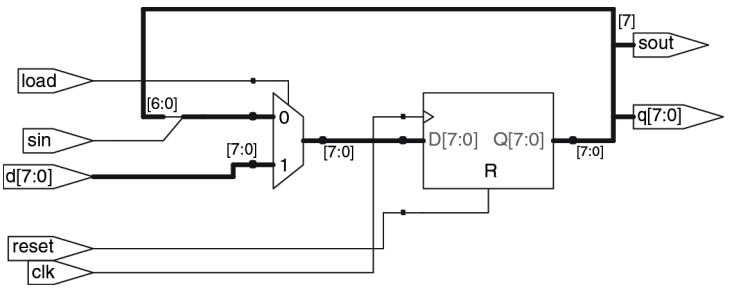


Рис. 5.38 Сдвиговый регистр с параллельной загрузкой

Сканирующие цепочки

Часто для тестирования последовательностных схем применяются *сканирующие цепочки* (*scan chains*), в которых используются сдвиговые регистры. Тестирование комбинационных схем производится относительно просто. На вход схемы подают специально подобранные входные сигналы, которые называются тестовыми векторами, а значения выходных сигналов сравнивают с ожидаемыми результатами. Тестирование последовательностных схем гораздо сложнее, поскольку их состояние зависит от предыстории входных сигналов. Если начальное состояние схемы зафиксировано, то для достижения интересующего состояния может потребоваться большое количество тестовых векторов. Например, для проверки корректности работы старшего разряда 32-битного счетчика необходимо сбросить счетчик, а затем подать на него 2^{31} (около двух миллиардов) тактовых импульсов!

> Не следует путать сдвиговые регистры и схемы сдвига, которые были рассмотрены в разделе 5.2.5. Сдвиговый регистр является последовательной схемой, в которую по каждому фронту тактового сигнала поступает новый бит. Схема сдвига является комбинационной схемой, которая сдвигает биты входного сигнала на указанную величину.

Для решения этой проблемы желательно иметь возможность непосредственно наблюдать и изменять все состояния схемы. Это достигается введением специального тестового режима, в котором содержимое всех триггеров может быть считано или изменено надлежащим образом. Большинство реальных систем содержат чрезвычайно много триггеров, поэтому невозможно выделить специальные контакты для чтения и изменения их содержимого. Вместо этого все триггеры системы соединены между собой в один огромный сдвиговый регистр, который называется сканирующей цепочкой. При нормальной работе триггеры получают данные со своих информационных входов D, а сканирование отключено. В тестовом режиме происходит последовательный сдвиг содержимого всех триггеров, которые входят в сканирующую цепочку: их старое содержимое поступает на выход S_{out}, а новое загружается через вход S_{in}. В состав *сканируемого триггера* (*scannable flip-flop*) кроме собственно триггера входит мультиплексор загрузки. На **рис. 5.39** приведена схема и графическое обозначение сканируемого триггера и показано, как триггеры соединяются последовательно для создания N-битного сканируемого регистра.

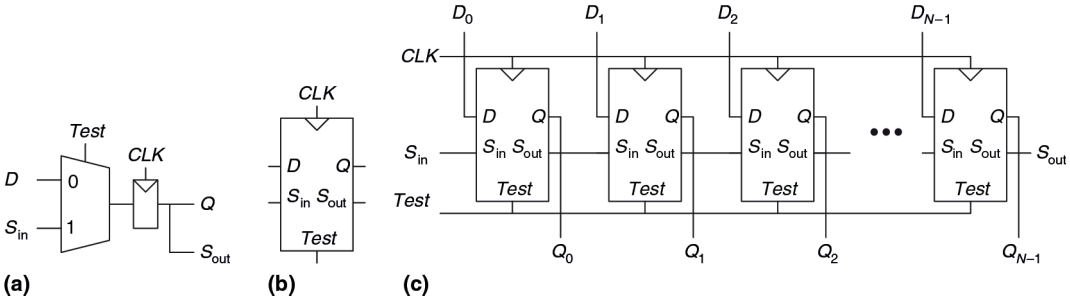


Рис. 5.39 Сканируемый триггер: (a) схема, (b) условное обозначение, (c) *N*-битный сканируемый регистр

Например, работу старшего разряда 32-битного счетчика можно протестировать следующим образом: в тестовом режиме он переводится в состояние 011111...111, затем выполняется один цикл счета в нормальном режиме, после этого в тестовом режиме считывается состояние счетчика, которое должно быть 100000...000. Эта последовательность действий требует только 32 + 1 + 32 = 65 циклов.

5.5. Матрицы памяти

В предыдущих разделах мы познакомились с арифметическими и последовательными схемами, которые используются для обработки данных. Для хранения этих данных и результатов работы схем в цифровых системах необходимы *запоминающие устройства* (*memories*). Регистр, состоящий из нескольких триггеров, является таким запоминающим устройством, предназначенным для хранения небольших объемов данных. В этом разделе мы рассмотрим *матрицы памяти*, которые позволяют эффективно хранить большие объемы данных.

Вначале мы познакомимся с общими характеристиками всех типов матриц памяти. Затем рассмотрим три типа матриц памяти: *динамическое оперативное запоминающее устройство* (ОЗУ, DRAM, динамическая память с произвольным доступом), *статическое оперативное запоминающее устройство* (СОЗУ, SRAM, статическая память с произвольным доступом), *постоянное запоминающее устройство* (ПЗУ, ROM, память только для чтения). Эти типы матриц отличаются способом хранения данных. Далее будут кратко проанализированы аппаратные затраты для создания матрицы памяти и их быстродействие. В конце раздела мы рассмотрим использование матриц памяти для выполнения функций комбинационной логики и способы их описания с помощью языков описания аппаратуры (HDL).

5.5.1. Обзор матриц памяти

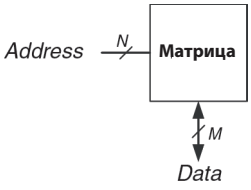


Рис. 5.40 Условное обозначение обобщенной матрицы памяти

На **рис. 5.40** показано графическое обозначение обобщенной матрицы памяти. Память организована как двумерная матрица запоминающих элементов. Содержимое памяти записывается и считывается по строкам. Строка выбирается *адресом* (*Address*). Записанные или считанные значения называются *данными* (*Data*). Матрица с N-битным адресом и M-битными данными имеет 2^N строк и M столбцов. Каждая строка данных называется *словом*. Таким образом, матрица содержит 2^N M-битных слов.

На **рис. 5.41** показана матрица памяти, адрес которой состоит из двух бит, а данные – из трех. Два адресных бита выбирают одну из четырех

строк (слов данных) матрицы. Ширина каждого слова данных равна трем битам. На **рис. 5.41 (b)** приведен пример возможного содержимого матрицы памяти. *Глубина* матрицы равна количеству ее строк, а ее *ширина* – количеству столбцов, которое также называется размером слова. Размер матрицы равен произведению количества столбцов на количество строк. На **рис. 5.41** показана матрица на 4 слова × 3 бита, или просто 4×3. Обозначение матрицы на 1024 слова × 32 бита показано на **рис. 5.42**. Общий размер этой матрицы равен 32 килобита (Кбит).

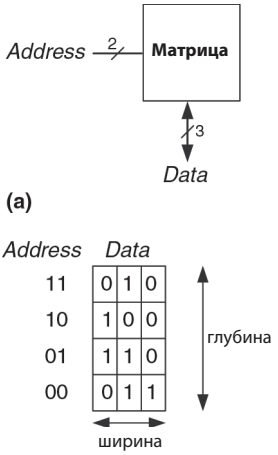


Рис. 5.41 Матрица памяти 4 × 3: (a) условное обозначение, (b) функция

Запоминающие элементы

Матрицы памяти представляют собой набор запоминающих элементов, каждый из которых хранит один бит данных. На **рис. 5.43** показано, что каждый запоминающий элемент соединен с линией слов (линией выборки слов) и линией битов (линией записи-считывания). При любой комбинации адресных битов активируется только одна линия выборки слов, и тем самым разрешается доступ к элементам соответствующей строки. Когда линия выборки слов некоторой строки активна, элементы этой строки могут выдавать данные на линии записи/чтения или принимать данные с этих линий. В противном случае запоминающие элементы отсоединены от линии записи/чтения. Для разных типов памяти схемы запоминающих элементов будут разными.

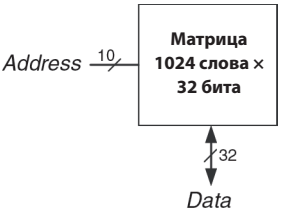


Рис. 5.42 Матрица 32 Кб: глубина = 2^{10} = 1024 слова, ширина = 32 бита

При чтении битов линия записи/чтения вначале находится в отключенном состоянии (Z). Затем включается линия выборки слов, и запоминающие элементы выдают хранимое значение на линию записи/чтения. При записи информации в запоминающий элемент сигнал на линию записи/чтения поступает со специального усилителя записи/чтения, имеющего небольшое выходное сопротивление. Затем включается линия выборки слов, и линии записи/чтения соединяются с запоминающими элементами. Сигнал с линии записи/чтения подавляет содержимое запоминающего элемента, и в элемент записывается новая информация.

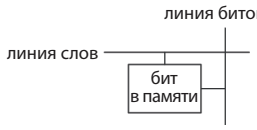


Рис. 5.43 Запоминающий элемент

Организация матрицы памяти

На **рис. 5.44** показана внутренняя организация матрицы памяти 4×3. Реальные запоминающие устройства имеют намного больший объем, но поведение малых матриц памяти может быть экстраполировано

на поведение больших. В этом примере матрица хранит данные, которые приведены на **рис. 5.41 (b)**.

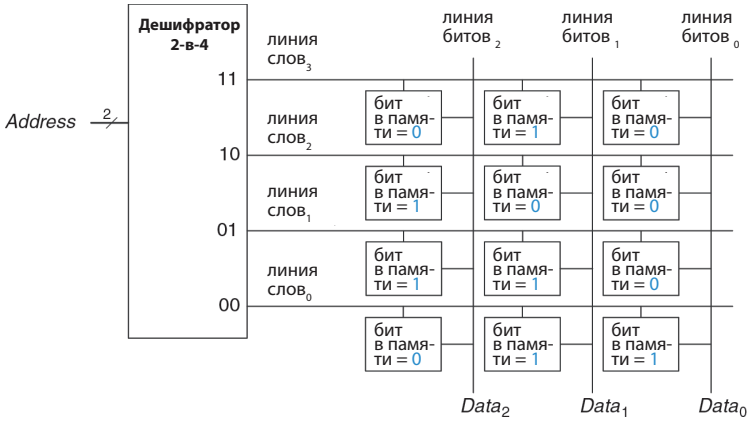


Рис. 5.44 Матрица памяти 4×3

При чтении содержимого памяти активируется линия выборки слов, и с запоминающих элементов соответствующей строки на линии записи/чтения поступает напряжение высокого или низкого логического уровня. При записи на линии записи/чтения с помощью усилителя записи/чтения подаются данные, которые будут сохранены в элементах строки, а затем активируется соответствующая линия выборки слов. Например, для чтения данных по адресу 10 линии записи/чтения остаются в отключенном состоянии, дешифратор активирует вторую линию выборки слов, и данные, которые хранятся в этой строке (100), считываются с линий записи/чтения (*Data*). Для записи значения 001 по адресу 11 на линии записи/чтения с усилителя записи/чтения поступает величина 001, затем активируется третья линия выборки слов, и новое значение сохраняется в запоминающих элементах.

Порты памяти

Память всех типов имеет один или несколько *портов* (*ports*). Через порты осуществляется доступ к содержимому памяти по некоторому адресу для чтения, записи или чтения/записи. В предыдущем примере была рассмотрена однопортовая память.

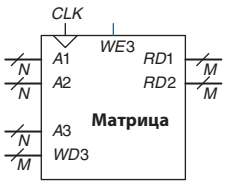


Рис. 5.45 Трехпортовая память

Многопортовая память обеспечивает одновременный доступ к содержимому по нескольким адресам. На **рис. 5.45** показана трехпортовая память с двумя портами для чтения и одним для записи. Порт 1 считывает данные, которые хранятся по адресу *A*1, и выдает их на выход *RD*1. Порт 2 выдает информацию, находящуюся по адресу *A*2, на выход *RD*2. Порт 3 позволяет записать данные, поданные на вход *WD*3,

в элемент по адресу *A*3, запись информации осуществляется по переднему фронту тактового импульса при активном сигнале *WE*3.

Типы памяти

Матрицы памяти характеризуются размером (глубина × ширина), количеством и типом портов. Память всех типов хранит данные в матрице запоминающих элементов, но способ хранения битов различный.

Запоминающие устройства классифицируются по способу хранения битов. Запоминающие устройства делятся на два больших класса: *оперативные запоминающие устройства* (ОЗУ, RAM, память с произвольным доступом) и *постоянные запоминающие устройства* (ПЗУ, ROM, память только для чтения). ОЗУ является энергозависимым, то есть при отключении питания информация, которая хранилась в ОЗУ, утрачивается. ПЗУ энергонезависимо, оно сохраняет свои данные даже при отсутствии питания.

Разделение запоминающих устройств на два больших класса – ОЗУ и ПЗУ – возникло на заре компьютерной эры и сейчас устарело и не отражает реальную ситуацию. В ОЗУ время доступа ко всем данным одинаково. Напротив, в запоминающих устройствах с последовательным доступом, таких как память на магнитной ленте, доступ к «ближним» данным происходит намного быстрее, чем доступ к «дальним» (например, тем, которые хранятся на противоположном конце магнитной ленты). Исторически ПЗУ называется постоянным, поскольку данные из такого устройства можно было считывать, но нельзя было записывать в него. Тем не менее в современные ПЗУ данные могут быть записаны. Главное отличие, на которое следует обратить внимание, состоит в том, что ОЗУ энергозависимо, а ПЗУ энергонезависимо.

Основными классами ОЗУ являются *динамическое* оперативное запоминающее устройство (динамическая память, *DRAM*) и *статическое* оперативное запоминающее устройство (статическая память, *SRAM*). Динамическая память сохраняет данные в виде заряда конденсаторов, а статическая – в виде состояния бистабильной схемы, состоящей из двух перекрестно соединенных инверторов. Существует много разновидностей ПЗУ, которые отличаются методами записи и считывания информации. Разные типы запоминающих устройств будут рассмотрены в следующих разделах.

Роберт Деннард, 1932 г. р.

Динамическое ОЗУ было изобретено в 1966 году на фирме IBM Робертом Деннардом. Хотя многие относились скептически к принципу работы динамического ОЗУ, с середины 1970-х годов динамическая память используется практически во всех компьютерах. По утверждению Деннарда, он мало занимался творческой работой до прихода в IBM, где руководство поручило ему задокументировать свои идеи и оформить на них патенты. После 1965 года он получил 35 патентов в области полупроводниковой техники и микроэлектроники. (Фотография любезно предоставлена IBM.)

5.5.2. Динамическое ОЗУ (DRAM)

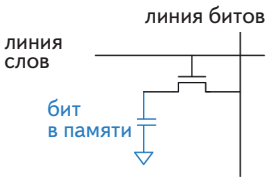


Рис. 5.46 Запоминающий элемент динамического ОЗУ

В *динамическом ОЗУ* (*DRAM*) значениям битов соответствует наличие и отсутствие заряда конденсатора. На **рис. 5.46** показан запоминающий элемент динамического ОЗУ. Значение бита сохраняется в конденсаторе. *N*-канальный МОП-транзистор (nMOS) является ключом, который может подключить конденсатор к линии записи/чтения или отключить его. Когда линия выборки слов активна, транзистор включается, и хранимые биты передаются на линию записи/чтения или наоборот, происходит запись новой информации в элемент.

Как показано на **рис. 5.47 (a)**, когда конденсатор заряжен до V_{DD}, хранимый бит равен 1; когда он разряжен до нуля (**рис. 5.47 (b)**), хранимый бит равен 0. Узел конденсатора будет динамическим, поскольку он фактически не управляется транзистором, подсоединенным к V_{DD} или GND.

При чтении данные передаются от конденсатора на линию записи/чтения. При записи данные поступают с линии записи/чтения на конденсатор. Чтение уничтожает данные, которые хранились в конденсаторе, поэтому после каждого чтения данные должны быть восстановлены (перезаписаны). Даже если из динамического ОЗУ не нужно считывать данные, из-за саморазряда конденсаторов они должны регенерироваться (считываться и перезаписываться) каждые несколько миллисекунд.

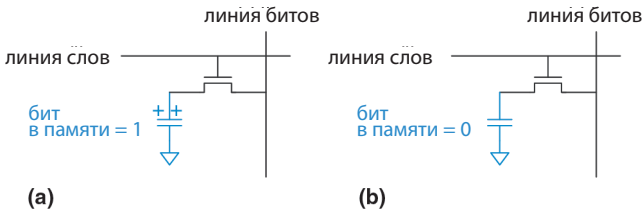


Рис. 5.47 Хранение данных в динамическом ОЗУ

5.5.3. Статическое ОЗУ (SRAM)

Статическое ОЗУ (*SRAM*) называется статическим, потому что в нем отсутствует необходимость регенерации хранимых данных. На **рис. 5.48** показан запоминающий элемент статического ОЗУ. Данные хранятся в бистабильной схеме, состоящей их двух перекрестно соединенных инверторов, подобной тем, которые были рассмотрены в **разделе 3.2**. Каждый запоминающий элемент имеет два выхода, *линия битов* и *линия битов*. Когда линия выборки слов активна, оба *n*-канальных МОП-транзистора открываются и данные могут быть записаны в эле-

мент или считаны из него. В отличие от динамического ОЗУ, перекрестно соединенные инверторы возвращают запоминающий элемент в равновесное состояние, если он из него выйдет вследствие случайных отклонений.

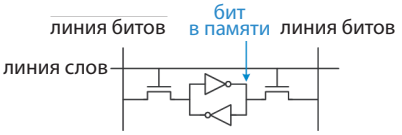


Рис. 5.48 Запоминающий элемент статического ОЗУ

5.5.4. Площадь и задержки

Триггеры, а также статические и динамические ОЗУ являются энергозависимыми запоминающими устройствами, но они различаются временными характеристиками и площадью чипа, необходимой для хранения одного бита. В **табл. 5.6** приведено сравнение этих трех типов энергозависимой памяти. Данные, хранимые в триггере, непосредственно доступны на его выходе. Но схема триггера состоит, по крайней мере, из 20 транзисторов. В общем случае чем больше транзисторов используется в схеме, тем большую площадь она занимает, потребляет больше энергии и стоит дороже. Задержка в динамическом ОЗУ больше, чем в статическом ОЗУ, потому что в нем линия записи/чтения фактически не управляется транзистором. Задержка динамического ОЗУ ограничивается относительно медленной передачей заряда из конденсатора на линию чтения/записи. Из-за необходимости выполнения периодической регенерации и регенерации после чтения динамическое ОЗУ имеет меньшую пропускную способность, чем статическое. Современные разновидности динамического ОЗУ, такие как *синхронное динамическое ОЗУ (SDRAM)* и *синхронное динамическое ОЗУ с удвоенной скоростью обмена (DDR SDRAM*, или коротко *DDR)* были разработаны для преодоления этой проблемы. В синхронном динамическом ОЗУ используется тактовый сигнал для конвейеризации доступа к памяти. В синхронном динамическом ОЗУ с удвоенной скоростью обмена передача данных происходит как по переднему, так и по заднему фронту тактового импульса, что удваивает пропускную способность при заданной частоте тактового

Таблица 5.6 Сравнение типов памяти

Тип памяти	Количество транзисторов в запоминающем элементе	Задержка
Триггер	~20	Малая
Статическое ОЗУ	6	Средняя
Динамическое ОЗУ	1	Большая

сигнала. Синхронное динамическое ОЗУ с удвоенной скоростью обмена было впервые стандартизировано в 2000 году и работало на частотах от 100 до 200 МГц. В более новых стандартах, DDR2, DDR3 и DDR4, тактовая частота была увеличена, и к 2012 году она превысила 1 ГГц.

Задержка памяти и ее пропускная способность также зависят от размера памяти; при прочих равных условиях память большего объема, как правило, работает медленнее, чем меньшего. Выбор лучшего типа памяти для конкретного проекта зависит от требований к быстродействию, цене и энергопотреблению.

5.5.5. Регистровые файлы

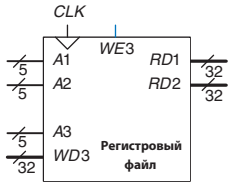


Рис. 5.49 Регистровый файл 32×32 с двумя портами чтения и одним портом записи

Цифровые системы часто используют несколько регистров для хранения временных переменных. Такие группы регистров, которые называются *регистровыми файлами*, обычно реализуются в виде небольших многопортовых матриц статического ОЗУ, поскольку они более компактны, чем матрицы триггеров.

На **рис. 5.49** показан трехпортовый регистровый файл, состоящий из 32 регистров по 32 бита каждый, который построен на основе трехпортовой памяти, подобной приведенной на **рис. 5.46**. Регистровый файл имеет два порта для чтения (*A1*/*RD1* и *A2*/*RD2*) и один порт для записи (*A3*/*WD3*). Пятиразрядные адреса *A1*, *A2* и *A3* обеспечивают доступ к любому из $2^5 = 32$ регистров. Таким образом, одновременно можно записывать информацию в один регистр и считывать из двух.

5.5.6. Постоянное запоминающее устройство

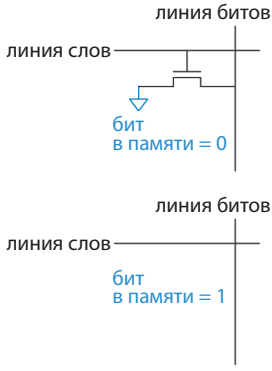


Рис. 5.50 Запоминающие элементы ПЗУ, содержащие 0 и 1

В *постоянном запоминающем устройстве* (*ПЗУ*, *ROM*) хранимым битовым значениям соответствует наличие или отсутствие транзистора. На **рис. 5.50** показан простой запоминающий элемент ПЗУ. При чтении информации из элемента на линию записи/чтения от внешнего источника подается уровень слабой логической 1. Затем активируется линия выборки слов. Если в элементе есть транзистор, он открывается и устанавливает на линии записи/чтения уровень логического 0. Когда транзистор отсутствует, на линии записи/чтения остается уровень логической 1. Обратите внимание на то, что ПЗУ является комбинационной схемой и не имеет состояния, которое может быть потеряно при отключении питания.

Содержимое ПЗУ может быть показано с помощью точечной нотации. На **рис. 5.51** приведена точечная нотация для

ПЗУ на 4 слова × 3 бита, которая содержит данные **рис. 5.41**. Наличие точки на пересечении строки (линии выборки слов) и столбца (линии записи/чтения) показывает, что хранимый бит равен 1. Например, на верхней линии выборки слов есть только одна точка на ее пересечении с $Data_1$, следовательно, по адресу 11 хранится значение 010.

Концептуально ПЗУ может быть построено с использованием двухуровневой логики, состоящей из группы логических элементов И, за которой следует группа элементов ИЛИ. Элементы И порождают все возможные минтермы и, следовательно, формируют дешифратор. На **рис. 5.52** показано ПЗУ **рис. 5.51**, построенное с использованием дешифратора и элементов ИЛИ. Каждая точка на **рис. 5.51** соответствует соединению строки и входа элемента ИЛИ на **рис. 5.52**.

Для выходных битов данных с одной точкой, таких как $Data_0$, элемент ИЛИ не нужен. Такое представление ПЗУ показывает, что с помощью ПЗУ можно реализовать произвольную двухуровневую логическую функцию. Реальные ПЗУ состоят из транзисторов, а не логических элементов, что позволяет уменьшить их размер и стоимость. В **разделе 5.6.3** реализация ПЗУ на уровне транзисторов будет рассмотрена детально.

Содержимое запоминающих элементов ПЗУ, которое показано на **рис. 5.50**, определяется при его изготовлении наличием или отсутствием транзистора в каждой ячейке. В *программируемом ПЗУ* (*ППЗУ*, *PROM*) транзисторы размещены во всех элементах, но в них есть возможность управлять соединением этих транзисторов с землей.

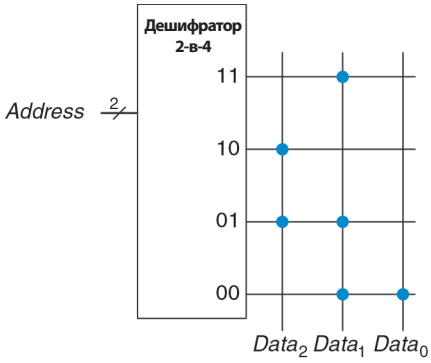


Рис. 5.51 ПЗУ 4 × 3: точечная нотация

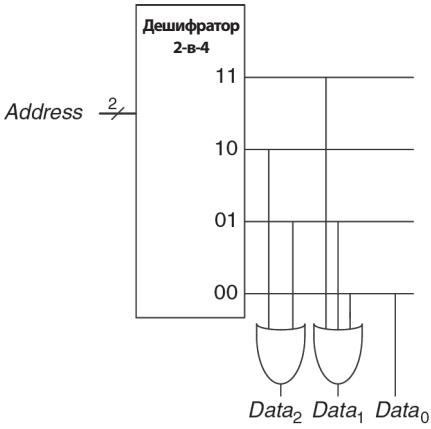


Рис. 5.52 Реализация ПЗУ 4 × 3 с использованием логических элементов

На **рис. 5.53** показан запоминающий элемент *ПЗУ, программируемого плавкими перемычками* (*fuse-programmable ROM*). Пользователь может программировать ПЗУ, подавая высокое напряжение на некоторые перемычки и тем самым пережигая их. Если перемычка присутствует, то транзистор соединен с землей, и элемент хранит 0. Если перемычка разрушена, то транзистор отсоединен от земли и элемент хранит 1. Такое ПЗУ также называют однократно программируемым ПЗУ, поскольку после пережигания перемычки ее невозможно восстановить.

В перепрограммируемых ПЗУ реализован механизм обратимого соединения-разъединения транзисторов с землей. В стираемых программируе-

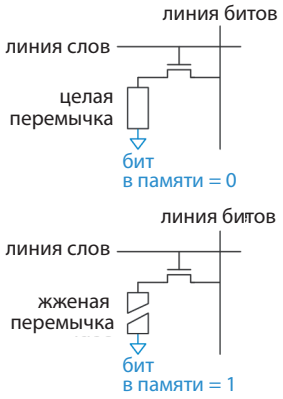


Рис. 5.53 Запоминающий элемент ПЗУ, программируемого перемычками

Фуджио Масуока, 1944 г. р.

Получил степень Ph. D. в области электротехники в университете Тохоку, Япония. Занимался разработкой запоминающих устройств и быстродействующих схем в фирме Toshiba с 1971 по 1995 год. Изобрел флеш-память в конце 1970-х годов при выполнении самостоятельного любительского проекта по ночам и выходным. Флеш-память получила свое имя из-за того, что процесс стирания памяти напоминает работу вспышки (flash) камеры. Toshiba запоздала с коммерческой реализацией идеи флеш-памяти; первенство принадлежит фирме Intel, которая предложила коммерческие изделия в 1988 году. Рынок флеш-памяти растет на $25 млрд в год. Доктор Масуока в дальнейшем присоединился к факультету университета Тохоку и работает над созданием трехмерного транзистора.

мых ПЗУ (*СППЗУ*, *erasable PROMs*, *EPROM*) *n*-МОП-транзисторы и перемычки заменены *транзисторами с плавающим затвором* (*floating-gate transistor*). Плавающий затвор не соединен физически ни с какими другими проводниками. Когда на транзистор подается достаточно высокое напряжение, электроны туннелируют через изолятор на плавающий затвор, транзистор включается и соединяет линию выборки слов и линию битов (выход дешифратора). Когда *СППЗУ* облучают ультрафиолетовым излучением в течение примерно получаса, электроны выбрасываются с плавающего затвора, и транзистор выключается. Эти действия называются *программированием* и *стиранием* соответственно. В *электрически стираемом программируемом ПЗУ* (*ЭСППЗУ*, *electrically erasable PROM*, *EEPROM*) и *флеш-памяти* (*flash memory*) используется аналогичный принцип, но ультрафиолетовое излучение не используется, поскольку на чипе присутствует специальная схема стирания. В ЭСППЗУ запоминающие элементы можно стирать индивидуально, во флеш-памяти стирание происходит большими блоками, она дешевле, поскольку в ней используется меньшее количество стирающих схем. В 2012 году стоимость флеш-памяти составляла примерно $1 за 1 Гб, и она продолжала снижаться примерно на 30–40 % за год. Флеш-память стала очень популярной для хранения больших объемов данных в переносных устройствах с питанием от батарейки, таких как камеры и музыкальные проигрыватели.

Таким образом, современные ПЗУ не являются постоянными в строгом значении этого слова: они могут программироваться, т. е. информация в них может записываться. Различие между ОЗУ и ПЗУ состоит в том, что запись в ПЗУ требует больше времени, и они являются энергонезависимыми.

5.5.7. Реализация логических функций с использованием матриц памяти

Хотя основным применением матриц памяти является хранение данных, они также могут использоваться для реализации комбинационных логических функций. На-

пример, выход $Data_2$ ПЗУ, которое показано на **рис. 5.51**, представляет собой функцию XOR двух входов *Address*. Аналогично $Data_0$ – это функция NOR двух входов. Матрица памяти размерностью 2^N слов $\times M$ бит может реализовать произвольную логическую функцию с N входами и M выходами. Например, ПЗУ на **рис. 5.51** реализует три функции двух аргументов.

Матрицы памяти, которые реализуют логические функции, называются *таблицами преобразований* (*look-up tables*, *LUT*). На **рис. 5.54** показана матрица памяти на 4 слова × 1 бит, которая используется как таблица преобразования для реализации функции $Y = AB$. При использовании памяти для выполнения логической функции для заданной комбинации входов (адреса) в ней происходит поиск соответствующего значения выхода. Каждый адрес соответствует строке в таблице истинности, а каждый хранимый бит – значению выходного сигнала.

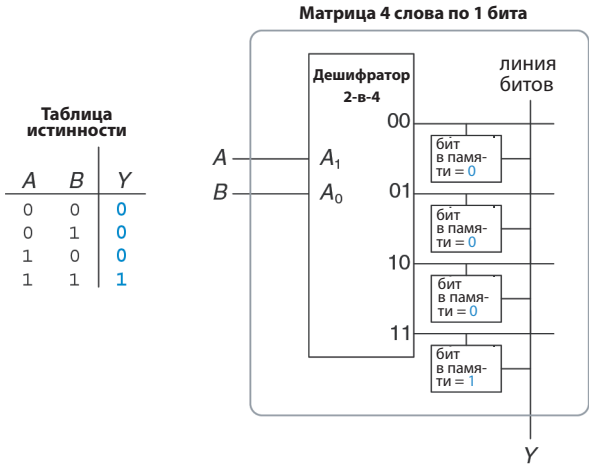
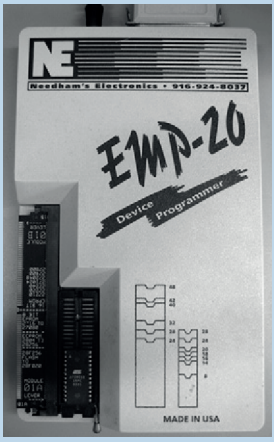


Рис. 5.54 Матрица 4 слова × 1 бит с использованием таблицы преобразования

Из-за быстрого падения цены накопители на основе флеш-памяти с разъемом USB заменили компакт-диски и дискеты.



Программируемые ПЗУ можно конфигурировать с помощью специального прибора – программатора, подобного показанному на рисунке. Прибор подсоединяется к компьютеру, который задает тип ПЗУ и данные, которые должны быть запрограммированы. Программатор пережигает перемычки или инжектирует заряд в плавающие затворы ПЗУ. Процесс программирования иногда называют прожиганием ПЗУ.

5.5.8. Языки описания аппаратуры и память

В **HDL-примере 5.6** на языках HDL описано ОЗУ размерностью 2^N слов $\times M$ бит. У этого ОЗУ есть синхронный вход разрешения записи. Другими словами, запись в память происходит по переднему фронту тактового импульса, если сигнал разрешения записи *we* (write enable) находится

в активном состоянии. Чтение происходит немедленно. Непосредственно после включения питания содержимое ОЗУ не определено.

HDL-пример 5.6 ОЗУ

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)

    if (we) mem [adr] <= din;
  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
       we:   in  STD_LOGIC;
       adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
       din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
       dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
   if rising_edge(clk) then
    if we then mem(TO_INTEGER(adr)) <= din;
    end if;
   end if;
  end process;

  dout <= mem(TO_INTEGER(adr));
end;
```

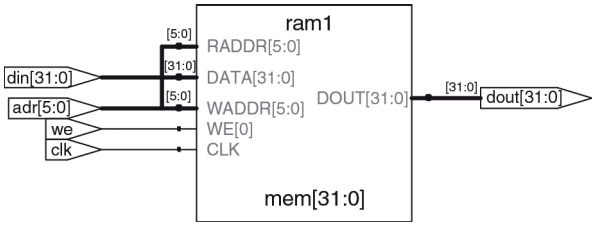


Рис. 5.55 ОЗУ

В **HDL-примере 5.7** приведено описание ПЗУ размером 4 слова × 3 бита. Содержимое ПЗУ задается в операторе `case`. Такое маленькое ПЗУ может быть синтезировано в виде набора логических элементов, а не матрицы. Напомним, что дешифратор семисегментного кода из **HDL-примера 4.24** был синтезирован в виде ПЗУ, приведенном на **рис. 4.20**. В **HDL-примере 5.8** описан 3-портовый регистровый файл 32×32 с нулевым входом, на который внутрисхемно подается фиксированное нулевое значение.

HDL-пример 5.7 ПЗУ

SystemVerilog

```
module rom(input logic  [1:0] adr,
           output logic [2:0] dout);

  always_comb
    case(adr)
      2'b00: dout = 3'b011;
     2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(all) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

HDL-пример 5.8 РЕГИСТРОВЫЙ ФАЙЛ

SystemVerilog

```
module regfile(input logic clk,
               input logic we3,
               input logic [5:0] a1, a2, a3,
               input logic [31:0] wd3,
               output logic [31:0] rd1, rd2);
  logic [31:0] rf[31:0];
  // трехпортовый регистровый файл
  // комбинационное чтение двух портов
  // (A1/RD1, A2/RD2)
  // запись в третий порт по переднему
  // фронту тактового импульса (A3/WD3/WE3)
  // значение регистра 0 жестко привязано
  // к значению 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity regfile is
  port(clk:        in  STD_LOGIC;
       we3:        in  STD_LOGIC;
       a1, a2, a3: in  STD_LOGIC_VECTOR(5  downto 0);
       wd3:        in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:   out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
                                    STD_LOGIC_VECTOR
                                      (31 downto 0);
  signal mem: ramtype;
begin
  -- трехпортовый регистровый файл
  -- комбинационное чтение двух портов (A1/RD1, A2/RD2)
  -- запись в третий порт по переднему фронту
  -- тактового импульса (A3/WD3/WE3)
  -- значение регистра 0 жестко привязано к 0
  process(clk) begin
    if rising_edge(clk) then
      if we3 = '1' then mem(to_integer(a3)) <= wd3;
      end if;
    end if;
  end process;
```

HDL-пример 5.8 (окончание)

```
  process(a1, a2) begin
    if (to_integer(a1) = 0) then rd1 <= X"00000000";
    else rd1 <= mem(to_integer(a1));
    end if;
    if (to_integer(a2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(to_integer(a2));
    end if;
  end process;
end;
```

5.6. Матрицы логических элементов

Логические элементы, как и запоминающие элементы, могут быть организованы в регулярные матрицы. Если соединения между логическими элементами программируемы, такие матрицы можно сконфигурировать для реализации произвольной логической функции, при этом не надо будет изменять соединения между микросхемами на плате. Регулярная структура упрощает проектирование. Матрицы логических элементов производятся в больших количествах, что обеспечивает их малую стоимость. Существует программное обеспечение, позволяющее перенести проекты цифровых устройств в такие матрицы. Большинство матриц логических элементов реконфигурируемые, что позволяет изменить проект без замены аппаратного обеспечения. Реконфигурируемость очень ценна при разработке и полезна при эксплуатации изделия, поскольку оно может быть обновлено путем простой загрузки новой конфигурации.

В этом разделе будут рассмотрены два типа матриц логических элементов: *программируемая логическая матрица*[1] (ПЛМ, programmable logic arrays, PLA) и *программируемая пользователем вентильная матрица* (ППВМ, field programmable gate arrays, FPGA). В программируемой логической матрице (ПЛМ), которая представляет собой более старую технологию, можно реализовать только комбинационные логические функции. Программируемая пользователем вентильная матрица (ППВМ) позволяет создавать как комбинационные, так и последовательные схемы.

5.6.1. Программируемые логические матрицы

Программируемые логические матрицы (*ПЛМ*, *PLA*) позволяют реализовать двухуровневые комбинационные логические схемы, заданные *совершенной дизъюнктивной нормальной формой* (СДНФ). На

[1] В отечественной литературе распространен термин «программируемая логическая интегральная схема» (ПЛИС) для всех программируемых матриц логических элементов.

рис. 5.56 показано, что ПЛМ состоит из матрицы И, за которой следует матрица ИЛИ. Входы (в прямой и инверсной формах) поступают на матрицу И, которая создает импликанты, которые, в свою очередь, объединяются функциями ИЛИ и формируют выходной сигнал матрицы. ПЛМ размерности $M \times N \times P$ бит имеет M входов, N импликантов и P выходов.

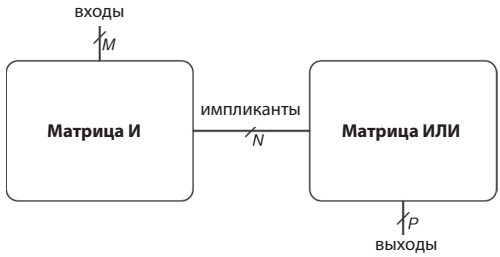


Рис. 5.56 Программируемая логическая матрица *M*×*N*×*P* бит

На **рис. 5.57** приведена точечная нотация ПЛМ 3×3×2 бит, которая реализовывает функции $X = \overline{A}\,\overline{B}C + AB\overline{C}$ и $Y = A\,\overline{B}$. Каждая строка в матрице И формирует импликант. Точки в строках матрицы И показывают, какие литералы формируют импликант.

Матрица И на **рис. 5.57** формирует три импликанта: $\overline{A}\,\overline{B}C + AB\overline{C}$ и $A\overline{B}$. Точки в матрице ИЛИ показывают, какие импликанты входят в выходную функцию.

На **рис. 5.58** проиллюстрировано, как ПЛМ может быть построена с использованием друхуровневой логики. Альтернативная реализация будет рассмотрена в **разделе 5.6.3**.

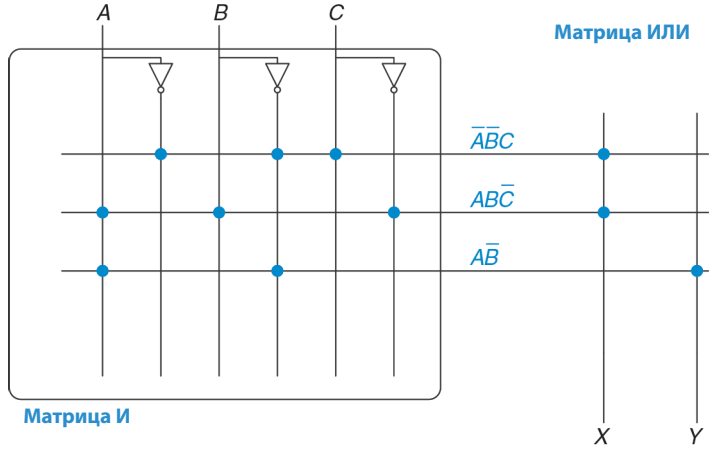


Рис. 5.57 Программируемая логическая матрица 3×3×2 бит: точечная нотация

ПЗУ можно рассматривать как разновидность ПЛМ. ПЗУ с организацией 2^M слов × N бит представляет собой ПЛМ-размерности $M \times 2^M \times N$ бит.

Дешифратор выполняет функции матрицы И и создает все 2^M минтермов. Массив запоминающих элементов выполняет функции матрицы ИЛИ и определяет выходные сигналы. Если функция зависит не от всех 2^M минтермов, то, весьма вероятно, реализация с ПЛМ будет более компактной, чем с ПЗУ. Например, для выполнения функций ПЛМ размерности 3×3×2 бит, которая показана на **рис. 5.57** и **5.58**, потребуется ПЗУ 8 слов × 2 бита.

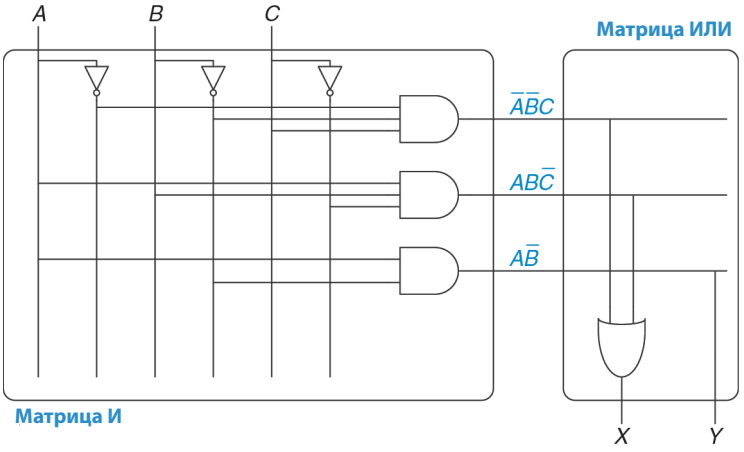


Рис. 5.58 Реализация программируемой логической матрицы 3×3×2 бит с использованием двухуровневой логики

В *простых программируемых логических устройствах* (*ППЛУ, SPLD*) базовые матрицы И и ИЛИ ПЛМ дополнены регистрами и дополнительными схемами. Однако в настоящее время ППЛУ и ПЛМ в основном вытеснены программируемыми пользователем вентильными матрицами (ППВМ), которые более гибки и эффективны при создании больших систем.

5.6.2. Программируемые пользователем вентильные матрицы

Программируемые пользователем вентильные матрицы (*ППВМ*, *FPGA*)[1] представляют собой матрицу реконфигурируемых элементов. С использованием специального программного обеспечения пользователь может описать свой проект на языке описания аппаратуры или в виде схемы, а затем реализовать его в FPGA. В ряде отношений матрицы FPGA мощнее и гибче, чем ПЛМ. В FPGA возможно реализовать как комбинационные, так и последовательностные схемы. В них можно реализовывать

[1] С этого места термин ПЛИС будет использоваться как синоним для ППВМ (FPGA).

многоуровневые логические схемы, тогда как в ПЛМ могут быть реализованы только двухуровневые схемы. В современные FPGA интегрированы другие полезные узлы, такие как умножители, высокоскоростные устройства ввода/вывода, ЦАП, АЦП, большие ОЗУ и процессоры.

FPGA представляет собой матрицу конфигурируемых *логических элементов* (*ЛЭ*, *logic elements*, *LE*), которые также называются *конфигурируемыми логическими блоками* (*КЛБ*, *configurable logic blocks*, *CLB*). Каждый ЛЭ можно сконфигурировать для выполнения функций некоторой комбинационной или последовательностной схемы. На **рис. 5.59** приведена обобщенная структура FPGA. ЛЭ окружены *элементами ввода/вывода* (*ЭВВ*, *input/output elements*, *IOE*), которые предназначены для организации обмена информацией между FPGA и прочими компонентами системы. Элементы ввода/вывода соединяют входы и выходы логических элементов с контактами корпуса микросхемы. Логические элементы могут быть соединены между собой и с элементами ввода/вывода с помощью программируемых каналов трассировки.

FPGA используются во многих потребительских продуктах, таких как автомобили, медицинское оборудование, устройства обработки медиаинформации. Например, в системах навигации, круиз-контроля, звуковоспроизведения автомобилей Mercedes Benz S-класса используется более десяти FPGA и PLD фирмы Xilinx. FPGA позволяют быстрее выводить изделия на рынок и упрощают отладку и добавление новых возможностей на поздних этапах жизненного цикла продукта.

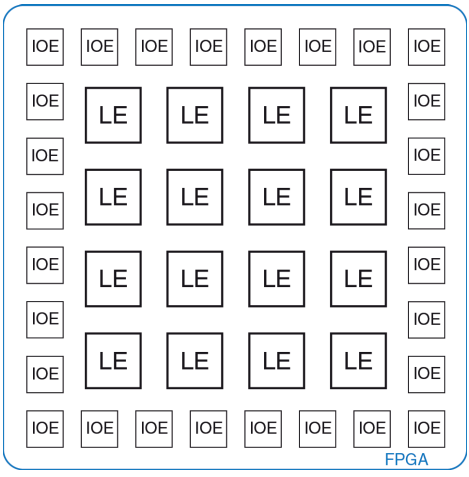


Рис. 5.59 Обобщенная структура FPGA

Лидерами на рынке FPGA являются фирмы Intel FPGA (ранее Altera Corp.) и Xilinx, Inc. На **рис. 5.60** показан один логический элемент схемы FPGA фирмы Altera Cyclone IV, производство которой началось в 2009 году. Основными компонентами логического элемента является четырехвходовая таблица преобразования (LUT) и однобитный регистр. Логический элемент также содержит конфигурируемые мультиплексоры, предназначенные для коммутации сигналов в логическом элементе. При программировании FPGA устанавливается содержимое таблиц пре-

образования (LUT) и определяются входные сигналы мультиплексоров, которые проходят на их выходы.

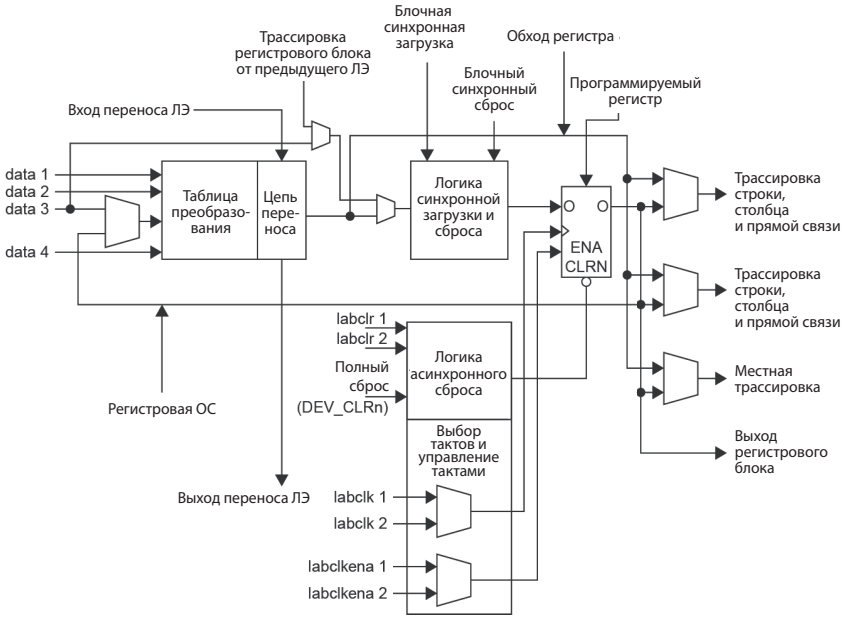


Рис. 5.60 Cyclone IV Logic Element (LE)
(воспроизведено с разрешения Altera Cyclone™ IV Handbook
© 2010 Altera Corporation)

Логический элемент FPGA Cyclone IV содержит одну четырехвходовую таблицу преобразования (LUT) и один триггер. Путем загрузки соответствующих значений в LUT она может быть сконфигурирована для реализации произвольной логической функции четырех (или менее) аргументов. Также при конфигурировании FPGA сигналами выбора, которые определяют, как мультиплексоры будут коммутировать каналы передачи данных в пределах логического элемента (LE) и между ним и соседними логическими элементами (LE) или элементами ввода/вывода (IOE), присваиваются необходимые значения. Например, в зависимости от конфигурации мультиплексора на один из входов LUT некоторого LE может поступать сигнал или с его входа *data 3*, или с выхода регистра этого же LE. На остальные три входа LUT сигналы всегда поступают со входов LE *data 1*, *data 2* и *data 4*. В зависимости от трассировки внешних соединений сигнал на входы *data 1–4* поступает с IOE или выходов других LE. Выход LUT может поступать либо непосредственно на выход LE при реализации комбинационной логической схемы, либо через триггер при создании последовательностной схемы. Сигнал на вход триггера может поступать с выхода LUT этого же LE, входа *data 3* или с выхода

регистра предыдущего LE. Кроме того, в LE входит ряд вспомогательных схем: дополнительные мультиплексоры для трассировки, схемы управления сигналами разрешения и сброса триггера, схемы, позволяющие реализовать сумматор с последовательным переносом. В FPGA фирмы Altera группы из 16 LE объединены в *блок логических матриц* (*logic array block*, *LAB*), для передачи данных между LE одного блока существуют специальные локальные соединения.

Таким образом, в LE FPGA Cyclone IV можно реализовать одну функцию четырех (или менее) входов, причем она может быть комбинационной или последовательностной, то есть иметь на выходе триггер. FPGA других производителей организованы немного по-другому, но принцип построения остается общим. Например, в FPGA фирмы Xilinx седьмой серии вместо четырехвходовый LUT используется шестивходовая.

При разработке конфигурации FPGA проектировщик вначале создает схемное описание проекта или описание на HDL. Затем происходит синтез проекта. Программный пакет для синтеза схем определяет, как следует сконфигурировать LUT, мультиплексоры и каналы трассировки для реализации заданных функций. Эта конфигурационная информация загружается в FPGA. Так как FPGA Cyclone IV сохраняют конфигурационную информацию в статическом ОЗУ, они могут быть легко перепрограммированы. Содержимое статического ОЗУ FPGA может быть загружено с компьютера (в лабораторных условиях) или при включении питания из специальной микросхемы ЭСППЗУ (EEPROM). Некоторые производители встраивают ЭСППЗУ непосредственно в микросхему FPGA или используют для конфигурирования FPGA однократно программируемые перемычки.

Пример 5.8 ПОСТРОЕНИЕ ФУНКЦИЙ С ИСПОЛЬЗОВАНИЕМ ЛОГИЧЕСКИХ ЭЛЕМЕНТОВ

Объясните, как следует сконфигурировать один или несколько логических элементов (LE) FPGA Cyclone IV для реализации следующих функций:

(a) $X = \overline{A}\,\overline{B}C + AB\overline{C}$ и $Y = A\overline{B}$;
(b) $Y = JKLMPQR$;
(c) счетчик по основанию 3 с двоичным кодированием состояния (**рис. 3.31 (a)**).

При необходимости вы можете показать связи между логическими элементами.

Решение (a) Для реализации функций следует сконфигурировать два логических элемента. Как показано на **рис. 5.61**, первая таблица преобразования (LUT) вычисляет *X*, вторая – *Y*. На входы *data 1*, *data 2* и *data 3* первой таблицы преобразования подаются сигналы *A*, *B* и *C* (эти соединения устанавливаются трассировочными каналами), вход *data 4* не используется, но на него нужно подать какое-либо значение, например 0. Во второй таблице преобразования на входы *data 1* и *data 2* подаются сигналы *A* и *B*; остальные входы не используются, и на них подан 0. Выходной мультиплексор сконфигурирован для подачи

на выход комбинационного сигнала с таблиц преобразования, таким образом на выходе формируются требуемые сигналы X и Y. В общем случае один логический элемент позволяет вычислить произвольную функцию четырех (или менее) аргументов.

(A) data 1	(B) data 2	(C) data 3	data 4	(X) выход ТП
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(A) data 1	(B) data 2	data 3	data 4	(Y) выход ТП
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

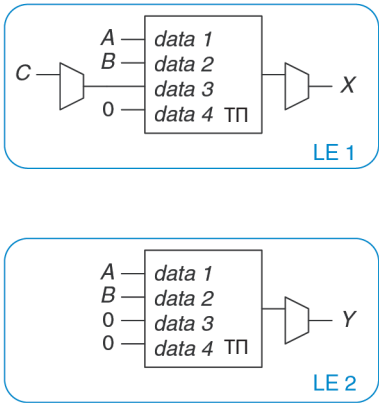


Рис. 5.61 Конфигурация логического элемента (LE) для реализации двух функций, имеющих до четырех входов

(b) Таблица преобразования (LUT) первого логического элемента (LE) должна быть сконфигурирована для вычисления $X = JKLM$, а второго – $Y = XPQR$. Выходные мультиплексоры должны выбирать комбинационные выходы X и Y каждого логического элемента (LE). Эта конфигурация показана на **рис. 5.62**. Трассировочные каналы между логическими элементами (LE), которые показаны синими пунктирными линиями, соединяют выход первого логического элемента со входом второго. В общем случае группа логических элементов (LE) позволяет вычислить аналогичным образом функцию N-входных переменных.

(c) Конечный автомат имеет два бита для хранения состояния ($S_{1:0}$) и один выход (Y). Следующее состояние зависит от двух битов текущего состояния. Как показано на **рис. 5.63**, для определения следующего состояния по текущему используется два логических элемента (LE). Два триггера, по одному из каждого логического элемента (LE), хранят это состояние. У триггеров есть вход сброса, который может быть соединен с внешним сигналом *Reset*. Синими пунктирными линиями показан тракт передачи сигнала через трассировочные каналы и мультиплексоры на входах *data 3* с выходных регистров назад на входы таблиц преобразования (LUT). В общем случае для вычисления выхода Y может понадобиться дополнительный логический элемент (LE). Но в данном случае $Y = S_0'$, то есть Y поступает с выхода первого логического элемента (LE). Таким образом, весь конечный автомат реализован на двух логических элементах (LE). В общем случае, для реализации конечного автомата необходимо по крайней мере по одному логическому элементу (LE) для каждого бита состояния; если логика определения выхода или следующего состояния слишком сложна для одной таблицы преобразования (LUT), то могут потребоваться дополнительные логические элементы (LE).

(J) data 1	(K) data 2	(L) data 3	(M) data 4	(X) выход ТП
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(P) data 1	(Q) data 2	(R) data 3	(X) data 4	(Y) выход ТП
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

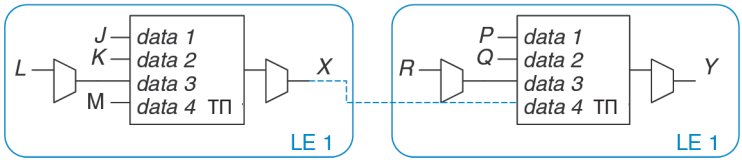


Рис. 5.62 Конфигурация логических элементов (LE) для реализации одной функции, имеющей более четырех входов

data 1	data 2	(S_0) data 3	(S_1) data 4	(S_0') выход ТП
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	data 2	(S_1) data 3	(S_0) data 4	(S_1') выход ТП
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

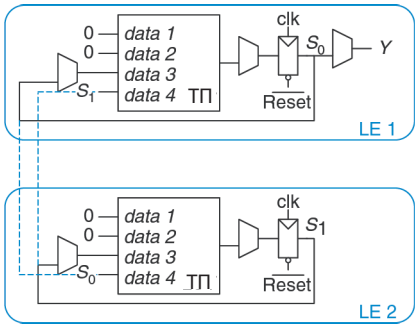


Рис. 5.63 Конфигурация логических элементов (LE) для реализации конечного автомата с состоянием, закодированным двумя битами

Пример 5.9 ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ЛОГИЧЕСКИХ ЭЛЕМЕНТОВ

Сколько требуется логических элементов (LE) Cyclone IV для реализации следующих схем?

(a) 4-входовый элемент AND.
(b) 7-входовый элемент XOR.
(c) $Y = A(B + C + D + E) + \overline{A}(BCDE)$.
(d) 12-разрядный сдвиговый регистр.

(e) 32-разрядный мультиплексор 2:1.
(f) 16-разрядный счетчик.
(g) Произвольный конечный автомат с 2 битами состояния, 2 входами и 3 выходами.

Решение

(a) 1: На основе LUT можно реализовать любую функцию до 4 входов.

(b) 2: первая LUT может вычислять XOR с четырьмя входами. Вторая LUT может вычислять XOR с еще тремя входами.

(c) 3: первая LUT вычисляет функцию четырех входов ($B + C + D + E$). Вторая LUT вычисляет BCDE – еще одну функцию четырех входов. Третья LUT использует 3 входа (два выхода предыдущих LUT и вход A) для вычисления Y.

(d) 12: регистру сдвига нужен один триггер на каждый разряд.

(e) 32: мультиплексор 2:1 является функцией трех входов: S, D_0 и D_1, поэтому для него требуется по одной LUT на каждый разряд.

(f) 16: для каждого разряда счетчика нужны триггер и полный сумматор. LE реализует логику триггера и сумматора. Хотя полный сумматор имеет два выхода и может показаться, что ему нужны два LUT, в LE есть специальная логика цепочки переноса (**рис. 5.60**), оптимизированная для выполнения сложения с одним LE.

(g) 5: конечный автомат состоит из двух триггеров, двух сигналов следующего состояния и трех выходных сигналов. Каждый следующий сигнал состояния является функцией четырех переменных (двух битов состояния и двух входов), поэтому его можно вычислить с помощью одной LUT. Таким образом, двух LE достаточно для логики вычисления следующего состояния и регистра состояний. Каждый выход является функцией максимум четырех сигналов, поэтому для каждого выхода требуется еще один LUT.

Пример 5.10 ЗАДЕРЖКА В ЛОГИЧЕСКОМ ЭЛЕМЕНТЕ

Алиса разрабатывает конечный автомат, который должен работать на частоте 200 МГц. Она использует FPGA Cyclone IV GX со следующими характеристиками: t_{LE} = 381 пс на LE, t_{setup} = 76 пс и t_{pcq} = 199 пс для всех триггеров. Задержка в соединении между LE равна 246 пс. Время удержания триггеров можно считать равным 0. Какое максимальное количество LE можно использовать в ее проекте?

Решение Для определения максимальной задержки распространения в комбинационной логической схеме Алиса использует **неравенство (3.13)**:

$$t_{pd} \le T_c - (t_{pcq} + t_{\text{setup}}).$$

Таким образом, t_{pd} = 5 нс – (0,199 нс + 0,076 нс), то есть $t_{pd} \le$ 4,725 нс. Задержка в каждом логическом элементе (LE) в сумме с задержкой в соединениях логических элементов ($t_{LE+wire}$) равна 381 пс + 246 пс = 627 пс. Максимальное количество (N) логических элементов (LE) можно определить из условия $Nt_{LE+wire} \le$ 4,725 нс. Таким образом, N = 7.

5.6.3. Схемотехника матриц

Во многих ПЗУ и ПЛМ вместо псевдо-*n*-МОП (pseudo-nMOS) используются динамические схемы. В динамических элементах *p*-МОП-транзистор включен не все время, что позволяет снижать энергопотребление. В остальных случаях динамические и псевдо-*n*-МОП-матрицы памяти похожи по схемотехнике и по режимам работы.

Для минимизации размеров и цены в ПЗУ и ПЛМ вместо традиционных логических элементов часто используются псевдо-*n*-МОП (pseudo-nMOS) или динамические (**раздел 1.7.8**) схемы.

На **рис. 5.64 (a)** представлена точечная нотация для ПЗУ 4×3 бит, которое реализует следующие функции: $X = A \oplus B$, $Y = \overline{A} + B$ и $Z = \overline{A + B}$. Это те же функции, которые были представлены на **рис. 5.51**, причем адресные входы были переобозначены как A и B, а выходы – X, Y и Z. Реализация с псевдо-*n*-МОП-элементами показана на **рис. 5.64 (b)**. Выход каждого дешифратора соединен с затворами *n*-МОП-транзисторов его строки. Как известно, в псевдо-*n*-МОП-схемах выход связан с цепью питания *p*-МОП-транзистором с большим сопротивлением канала. Выход имеет высокий потенциал, только если *n*-МОП-транзистор, который связывает его с землей, закрыт. Эти транзисторы расположены на всех пересечениях, где точка *отсутствует*. Для сравнения на **рис. 5.64 (b)** сохранены точки точечной нотации, которая была показана на **рис. 5.64 (a)**. *p*-МОП-транзисторы устанавливают высокий логический уровень всех линий слов, на которых *n*-МОП-транзисторы отсутствуют. Например, когда $AB = 11$, линия слов 11 имеет высокий уровень напряжения, соединенные с ней *n*-МОП-транзисторы открываются и на выходах X и Z устанавливают низкое напряжение. На пересечении линии выхода Y и линии 11 *n*-МОП-транзистор отсутствует, следовательно, на этом выходе сохраняется высокое напряжение.

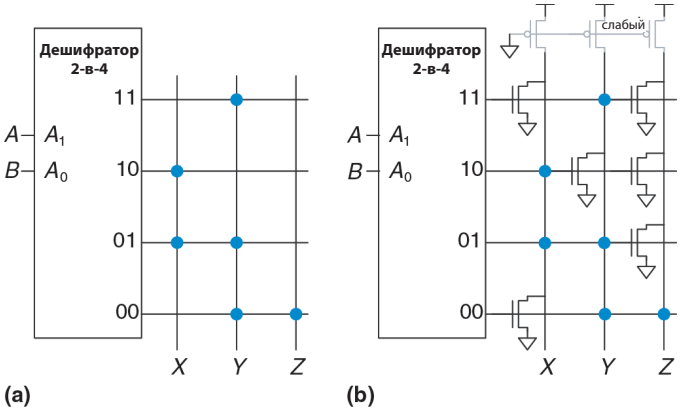


Рис. 5.64 Реализация ПЗУ: (a) точечная нотация, (b) псевдо-*n*-МОП-схема

ПЛМ также могут быть реализованы с использованием псевдо-*n*-МОП-схем. На **рис. 5.65** показана такая реализация ПЛМ, которая была изображена на **рис. 5.57**. *n*-МОП-транзисторы, обеспечивающие низкий логический уровень сигнала, расположены на *не отмеченных точками*

пересечениях матрицы И и в отмеченных строках матрицы ИЛИ. Столбцы матрицы ИЛИ поступают на выход через инверторы. Для сравнения синие точки с точечной нотации (**рис. 5.57**) показаны на **рис. 5.65**.

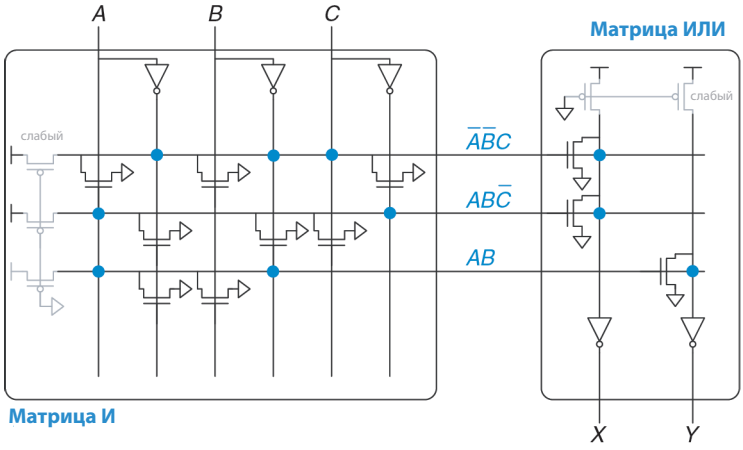


Рис. 5.65 Реализация ПЛМ 3×3×2 бит с использованием псевдо-*n*-МОП-схем

5.7. Заключение

В этой главе были рассмотрены функциональные узлы, которые используются во многих цифровых системах. В число таких функциональных узлов входят арифметические схемы: сумматоры, блоки вычитания, умножители, делители, схемы сдвига, последовательностные схемы: счетчики, сдвиговые регистры, логические матрицы и запоминающие устройства. В этой главе также были рассмотрены представления дробных чисел с плавающей и фиксированной запятыми. В **главе 7** эти функциональные узлы будут использоваться для построения микропроцессора.

Большое количество арифметических схем строятся с использованием сумматоров. Полусумматор имеет два однобитовых входа A и B и два выхода – сумма и перенос. В полном сумматоре ко входам полусумматора добавляется вход переноса. N полных сумматоров можно соединить последовательно и тем самым создать параллельный сумматор, который складывает два N-битовых числа. Такой сумматор также называют сумматором с последовательным переносом. Более быстрые параллельные сумматоры можно создать с использованием технологий группового ускоренного и префиксного переноса.

В блоке вычитания знак второго операнда инвертируется, а затем выполняется операция сложения. Схема сравнения вычитает одно число из другого, а результат сравнения определяется по знаку разницы. В умножителе элементы И формируют частичные произведения, а затем они складываются с помощью полных сумматоров. В схеме деления делитель многократно вычитается из частичного остатка, и по знаку разни-

цы определяются двоичные разряды частного. В счетчике для хранения состояния используется регистр, а для его увеличения – сумматор.

Дробные числа представляются в формах с плавающей или с фиксированной запятой. Представление с фиксированной запятой аналогично десятичному, а с плавающей – экспоненциальному. Для обработки чисел с фиксированной запятой используются обычные арифметические схемы, а числа с плавающей запятой требуют использования более сложных схем, которые выделяют и обрабатывают знак, порядок и мантиссу.

Запоминающие устройства большого объема организованы в виде матрицы слов. Запоминающие устройства имеют один или более портов для чтения и/или записи слов. Содержимое энергозависимой памяти, такой как статическое или динамическое ОЗУ, утрачивается при выключении питания схемы. Статическое ОЗУ быстрее, чем динамическое, но использует больше транзисторов. Регистровый файл представляет собой небольшое многопортовое статическое ОЗУ. Содержимое энергонезависимой памяти, которая называется постоянным запоминающим устройством (ПЗУ), сохраняется неограниченно долго при отсутствии питания. Несмотря на название, содержимое большинства современных ПЗУ может быть изменено.

Логические элементы также могут быть организованы в виде матриц. Для выполнения функций комбинационной логики могут использоваться матрицы памяти, в которых хранится таблица преобразования. ПЛМ состоит из соединенных между собой конфигурируемых матриц И и ИЛИ, в ПЛМ могут быть реализованы только комбинационные схемы. FPGA содержит большое количество небольших таблиц преобразования и регистров и позволяет реализовывать как комбинационные, так и последовательные схемы. Содержимое таблиц преобразования и их межсоединение могут быть сконфигурированы для выполнения любой логической функции. Современные FPGA могут быть легко перепрограммированы, содержат большое количество конфигурируемых логических элементов, весьма дешевы, что позволяет на их основе создавать сложные цифровые системы. Они широко используются как в коммерческих мало- и среднесерийных изделиях, так и в образовательных проектах.

Упражнения

Упражнение 5.1 Чему будет равна задержка следующих 64-разрядных сумматоров? Задержка любого двухвходового логического элемента равна 150 пс, а полного сумматора – 450 пс:

a) сумматор с последовательным переносом;
b) сумматор с ускоренным переносом, состоящий из 4-разрядных блоков;
c) префиксный сумматор.

Упражнение 5.2 Разработайте два сумматора с распространяющимся переносом: 64-разрядный сумматор с последовательным переносом и 64-разрядный сумматор с ускоренным переносом, состоящий из 4-разрядных блоков. Используйте только двухвходовые логические элементы. Каждый такой элемент имеет

площадь 15 мкм^2, задержку 50 пс и полную емкость 20 пФ. Статической мощностью можно пренебречь.

a) Сравните площадь, задержку и потребляемую мощность сумматоров, работающих на частоте 100 МГц при напряжении питания 1,2 В.
b) Обсудите компромисс между мощностью, площадью и задержкой.

Упражнение 5.3 Объясните, почему разработчик может использовать сумматор с последовательным переносом, а не сумматор с ускоренным переносом.

Упражнение 5.4 Разработайте 16-разрядный префиксный сумматор, показанный на **рис. 5.7**, с использованием языков описания аппаратуры. Проведите моделирование и тестирование своего модуля и покажите, что он работает корректно.

Упражнение 5.5 В префиксной сети, показанной на **рис. 5.7**, для вычисления всех префиксов используются черные ячейки. Сигналы распространения некоторых блоков на самом деле не нужны. Спроектируйте «серую ячейку», которая получает сигналы G и P для битов $i{:}k$ и $k{-}1{:}j$, но вычисляет только $G_{i:j}$, а не $P_{i:j}$. Перерисуйте префиксную сеть так, чтобы в ней везде, где возможно, черные ячейки были заменены на серые.

Упражнение 5.6 Префиксная сеть, показанная на **рис. 5.7**, – не единственный способ вычисления всех префиксов с логарифмической задержкой. Сеть Когге–Стоуна является другой распространенной префиксной сетью, которая выполняет те же функции с использованием иного соединения черных ячеек. Исследуйте сумматор Когге–Стоуна и нарисуйте схему, подобную показанной на **рис. 5.7**, на которой черные ячейки будут формировать сумматор Когге–Стоуна.

Упражнение 5.7 Вспомните, что N-входовый приоритетный шифратор имеет $\log_2 N$ выходов, на которых формируется двоичное число, соответствующее номеру самого старшего входа, на который подана логическая 1 (**упражнение 2.36**).

a) Разработайте N-входовый приоритетный шифратор, у которого задержка увеличивается логарифмически с ростом N. Нарисуйте схему шифратора и рассчитайте его задержку, исходя из задержек отдельных логических элементов.
b) Опишите ваш проект на языке описания аппаратуры. Проведите моделирование и тестирование своего модуля и покажите, что он работает корректно.

Упражнение 5.8 Разработайте следующие компараторы 32-разрядных чисел:

a) не равно;
b) больше, чем;
c) меньше или равно.

Нарисуйте их схемы.

Упражнение 5.9 Проанализируйте компаратор для сравнения чисел со знаком, показанный на **рис. 5.12**. Задание:

a) приведите пример двух 4-разрядных чисел со знаком A и B, для которых 4-разрядный компаратор правильно вычисляет $A < B$ с учетом знака;
b) приведите пример двух 4-разрядных чисел со знаком A и B, для которых 4-разрядный компаратор неправильно вычисляет $A < B$ с учетом знака;

c) в каких случаях N-разрядный компаратор со знаком работает неправильно? Сделайте обобщающий вывод.

Упражнение 5.10 Модифицируйте N-разрядный компаратор со знаком, показанный на **рис. 5.12**, чтобы он правильно выполнял сравнение $A < B$ для всех N-разрядных входных чисел A и B.

Упражнение 5.11 Разработайте 32-разрядное АЛУ, показанное на **рис. 5.15**, с использованием вашего любимого языка описания аппаратуры. Модуль верхнего уровня может быть или структурным, или поведенческим.

Упражнение 5.12 Разработайте 32-разрядное АЛУ, показанное на **рис. 5.17**, с использованием вашего любимого языка описания аппаратуры. Модуль верхнего уровня может быть или структурным, или поведенческим.

Упражнение 5.13 Разработайте 32-разрядное АЛУ, показанное на **рис. 5.18 (a)**, с использованием вашего любимого языка описания аппаратуры. Модуль верхнего уровня может быть разработан методом структурного или поведенческого описания.

Упражнение 5.14 Разработайте 32-разрядное АЛУ, приведенное на **рис. 5.18 (b)**, с использованием вашего любимого языка описания аппаратуры. Модуль верхнего уровня может быть разработан методом структурного или поведенческого описания.

Упражнение 5.15 Разработайте тестбенч для верификации 32-разрядного АЛУ из **упражнения 5.11** и выполните проверку. Разработайте все необходимые файлы с тестовыми векторами. Проведите моделирование работы схемы при граничных условиях.

Упражнение 5.16 Повторите **упражнение 5.15** для АЛУ из **упражнения 5.12**.

Упражнение 5.17 Повторите **упражнение 5.15** для АЛУ из **упражнения 5.13**.

Упражнение 5.18 Повторите **упражнение 5.15** для АЛУ из **упражнения 5.14**.

Упражнение 5.19 Разработайте блок беззнакового компаратора, который сравнивает два числа в формате без знака A и B. На вход модуля подается сигнал $Flags$ (N, Z, C, V) из АЛУ (**рис. 5.16**), выполняющего вычитание $A - B$. На выходе модуля появляется один из сигналов, означающих, что число A больше или равно (HS), меньше или равно (LS), больше (HI) или меньше (LO), чем B.

a) Найдите минимальные функции для вычисления HS, LS, HI и LO, исходя из N, Z, C и V.
b) Разработайте комбинационные схемы формирования сигналов HS, LS, HI и LO.

Упражнение 5.20 Разработайте модуль компаратора, который сравнивает два числа A и B в формате со знаком. На вход модуля подается сигнал $Flags$ (N, Z, C, V) из АЛУ (**рис. 5.16**), выполняющего вычитание $A - B$. На выходе модуля формируется один из сигналов, означающих, что число A больше или равно (GE), меньше или равно (LE), больше (GT) или меньше (LT), чем B.

a) Найдите минимальные функции для вычисления GE, LE, GT и LT, исходя из N, Z, C и V.

b) Разработайте комбинационные схемы формирования сигналов *GE*, *LE*, *GT* и *LT*.

Упражнение 5.21 Разработайте сдвиговый регистр, который сдвигает 32-битный вход влево на два бита. Выход также состоит из 32 бит. Сделайте словесное описание работы модуля и разработайте его схему. Реализуйте ваш проект с использованием вашего любимого языка описания аппаратуры.

Упражнение 5.22 Разработайте 4-разрядную схему циклического сдвига влево и вправо. Разработайте схему модуля. Реализуйте проект с использованием вашего любимого языка описания аппаратуры.

Упражнение 5.23 Спроектируйте 8-разрядную схему сдвига влево с использованием только 24 мультиплексоров 2:1. На вход схемы поступает 8-битный входной сигнал и 3-битная величина сдвига, $shamt_{2:0}$. На выходе схемы формируется 8-битный сигнал *Y*. Нарисуйте принципиальную схему.

Упражнение 5.24 Опишите, как построить любую схему *N*-разрядного сдвига или циклического сдвига, используя всего $N\log_2 N$ мультиплексоров 2:1.

Упражнение 5.25 Двухуровневая схема сдвига, приведенная на **рис. 5.66**, может выполнять любую *N*-разрядную операцию сдвига или цикличного сдвига. Она сдвигает $2N$-битный вход вправо на *k* бит. *N* младших бит результата поступают на выход *Y*. Старшие *N* бит входа обозначены через *B*, младшие *N* бит – через *C*. При соответствующем выборе *B*, *C* и *k* двухуровневая схема сдвига может выполнять любой сдвиг или циклический сдвиг. Опишите, как *B*, *C*, и *k* связаны с *A*, *shamt* и *N* для выполнения:

a) логического сдвига *A* вправо на *shamt*;
b) арифметического сдвига *A* вправо на *shamt*;
c) сдвига *A* влево на *shamt*;
d) циклического сдвига *A* вправо на *shamt*;
e) циклического сдвига *A* влево на *shamt*.

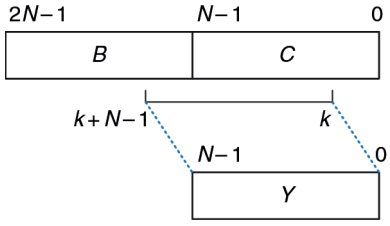


Рис. 5.66 Двухуровневая схема сдвига

Упражнение 5.26 Найдите критический путь и время прохождения сигнала по нему для умножителя 4×4, приведенного на **рис. 5.21**, считая известными задержки элемента И (t_{AND}) и сумматора (t_{FA}). Чему будет равна задержка аналогичного умножителя $N\times N$?

Упражнение 5.27 Найдите критический путь и время прохождения сигнала по нему для схемы деления 4×4, приведенной на **рис. 5.22**, считая известными задержки мультиплексора 2:1 (t_{MUX}), сумматора (t_{FA}) и инвертора (t_{INV}). Чему будет равна задержка аналогичной схемы деления $N\times N$?

Упражнение 5.28 Разработайте умножитель, который работает с числами, представленными в дополнительном коде.

Упражнение 5.29 Модуль расширения знака увеличивает количество разрядов числа, представленного в дополнительном коде, с M до N ($N > M$) путем копирования самого старшего бита входа в старшие биты выхода (**раздел 1.4.6**). Модуль имеет M-разрядный вход A и N-разрядный выход Y. Нарисуйте схему модуля расширения знака с 4-разрядным входом и 8-разрядным выходом. Реализуйте ваш проект на языке описания аппаратуры.

Упражнение 5.30 Модуль дополнения нулями увеличивает количество разрядов беззнакового числа с M до N ($N > M$) путем присвоения старшим битам выхода нулевого значения. Нарисуйте схему модуля дополнения нулями с 4-разрядным входом и 8-разрядным выходом. Опишите ваш проект на языке описания аппаратуры.

Упражнение 5.31 Вычислите $111001.000_2 / 001100.000_2$ в двоичной системе счисления, используя стандартный школьный алгоритм деления. Опишите процесс вычислений.

Упражнение 5.32 Числа какого диапазона можно представить с использованием следующих форматов:

a) формат U12.12 (24-битное беззнаковое число с фиксированной запятой с 12 битами целой части и 12 битами дробной части);
b) 24-битное число в прямом коде с фиксированной запятой с 12 битами целой части и 12 дробной;
c) формат Q12.12 (24-битное число в дополнительном коде с фиксированной запятой с 12 битами целой части и 12 битами дробной части).

Упражнение 5.33 Представьте следующие десятичные числа в 16-разрядном двоичном формате в прямом коде с 8 битами целой части и 8 битами дробной части. Выразите ответ в шестнадцатеричной системе счисления.

a) −13,5625.
b) 42,3125.
c) −17,15625.

Упражнение 5.34 Представьте следующие десятичные числа в 12-разрядном двоичном формате в прямом коде с 6 битами целой части и 6 битами дробной части. Выразите ответ в шестнадцатеричной системе счисления.

a) −30,5.
b) 16,25.
c) −8,078125.

Упражнение 5.35 Представьте десятичные числа из **упражнения 5.33** в формате Q8.8 (16-разрядный двоичный формат в дополнительном коде с 8 битами целой части и 8 битами дробной части). Выразите ответ в шестнадцатеричной системе счисления.

Упражнение 5.36 Представьте десятичные числа из **упражнения 5.34** в формате Q6.6 (12-разрядном двоичном формате в дополнительном коде с 6 битами целой части и 6 битами дробной части). Выразите ответ в шестнадцатеричной системе счисления.

Упражнение 5.37 Представьте десятичные числа из **упражнения 5.33** в формате с плавающей запятой и одинарной точностью в соответствии со стандартом IEEE 754. Выразите ответ в шестнадцатеричной системе счисления.

Упражнение 5.38 Представьте десятичные числа из **упражнения 5.34** в формате с плавающей запятой и одинарной точностью в соответствии со стандартом IEEE 754. Выразите ответ в шестнадцатеричной системе счисления.

Упражнение 5.39 Преобразуйте следующие числа в формате Q4.4 (двоичные числа с фиксированной запятой, заданные в дополнительном коде) в десятичные. Для простоты двоичная точка в этом примере показана явно.

a) 0101,1000.
b) 1111,1111.
c) 1000,0000.

Упражнение 5.40 Повторите **упражнение 5.39** для чисел в формате Q6.5 (двоичных чисел с фиксированной запятой, заданных в дополнительном коде).

a) 011101,10101.
b) 100110,11010.
c) 101000,00100.

Упражнение 5.41 При сложении двух чисел с плавающей запятой мантисса числа с меньшим порядком сдвигается. Зачем это делается? Опишите словесно и приведите пример, подтверждающий ваше объяснение.

Упражнение 5.42 Сложите следующие числа, заданные в формате с плавающей запятой и одинарной точностью в соответствии со стандартом IEEE 754.

a) C0123456 + 81C564B7.
b) D0B10301 + D1B43203.
c) 5EF10324 + 5E039020.

Упражнение 5.43 Сложите следующие числа, заданные в формате с плавающей запятой и одинарной точностью в соответствии со стандартом IEEE 754.

a) C0D20004 + 72407020.
b) C0D20004 + 40DC0004.
c) (5FBE4000 + 3FF80000) + DFDE4000.

(Почему полученные результаты интуитивно неочевидные? Объясните.)

Упражнение 5.44 Модифицируйте процедуру сложения чисел с плавающей запятой, описанную в **разделе 5.3.2**, для выполнения вычислений как с положительными, так и с отрицательными числами.

Упражнение 5.45 Рассмотрим числа, заданные в формате с плавающей запятой и одинарной точностью в соответствии со стандартом IEEE 754.

a) Сколько чисел можно представить в таком формате? Особые случаи $\pm\infty$ или NaN учитывать не нужно.
b) Сколько дополнительных чисел можно представить в данном формате, если не вводить в рассмотрение особые случаи $\pm\infty$ и NaN?
c) Поясните, почему для $\pm\infty$ и NaN выделено специальное представление.

Упражнение 5.46 Рассмотрим следующие десятичные числа: 245 и 0,0625.

a) Запишите эти числа в формате с плавающей запятой и одинарной точностью. Выразите ваш ответ в шестнадцатеричной системе счисления.
b) Выполните сравнение величин двух 32-разрядных чисел, полученных в задании (a). Другими словами, интерпретируйте два 32-разрядных числа как числа в дополнительном коде и сравните их. Будет ли сравнение таких целых чисел давать корректный результат?
c) Вы решили предложить новый формат с плавающей запятой и одинарной точностью. Единственное отличие от стандарта IEEE 754 чисел с плавающей запятой и одинарной точностью состоит в том, что вы предлагаете для порядка использовать дополнительный код, а не смещение. Запишите два числа в соответствии с вашим новым стандартом. Выразите ваш ответ в шестнадцатеричной системе счисления.
d) Будет ли целочисленное сравнение работать с новым форматом из задания (c)?
e) Почему удобно использовать алгоритм сравнения целых чисел для чисел с плавающей запятой?

Упражнение 5.47 Разработайте сумматор чисел с плавающей запятой и одинарной точностью с использованием вашего любимого языка описания аппаратуры. Перед разработкой кода нарисуйте схему вашего проекта. Промоделируйте и проведите тестирование вашего сумматора, чтобы доказать, что он работает корректно. Вы можете ограничиться использованием только положительных чисел и округление выполнять до нуля (выполнять усечение). Также вы можете не рассматривать особые случаи, приведенные в **табл. 5.4**.

Упражнение 5.48 Необходимо разработать 32-разрядный умножитель с плавающей запятой. Умножитель имеет два 32-битных входа для чисел с плавающей запятой и один 32-битный выход. Вы можете ограничиться использованием только положительных чисел и округление выполнять до нуля (выполнять усечение). Также вы можете не рассматривать особые случаи, приведенные в **табл. 5.4**.

a) Опишите последовательность шагов, необходимых для умножения 32-битных чисел с плавающей запятой.
b) Нарисуйте схему 32-разрядного умножителя с плавающей запятой.
c) Опишите 32-разрядный умножитель с плавающей запятой на языке описания аппаратуры. Промоделируйте и проведите тестирование вашего умножителя, чтобы доказать, что он работает корректно.

Упражнение 5.49 В этом упражнении вам нужно будет разработать 32-разрядный префиксный сумматор:

a) разработайте схему вашего проекта;
b) разработайте 32-разрядный префиксный сумматор с использованием языка описания аппаратуры. Промоделируйте и проведите тестирование вашего сумматора и докажите, что он работает корректно;
c) чему будет равна задержка 32-разрядного префиксного сумматора, спроектированного в задании (a)? Задержка каждого двухвходового логического элемента равна 100 пс;
d) разработайте конвейерную версию 32-битного префиксного сумматора, нарисуйте его схему. Насколько быстро будет работать конвейерный префиксный сумматор? Потери на упорядочение (t_{pcq} + t_{setup}) равны

80 пс. Спроектируйте сумматор так, чтобы он имел максимально возможное быстродействие;

e) разработайте 32-разрядный конвейерный префиксный сумматор с использованием языка описания аппаратуры.

Упражнение 5.50 Инкрементор к N-разрядному числу прибавляет 1. Постройте 8-разрядный инкрементор с использованием полусумматоров.

Упражнение 5.51 Постройте 32-разрядный *синхронный реверсивный счетчик* (Up/Down counter). Он имеет входы *Reset* и *Up*. Когда вход *Reset* установлен в 1, все выходы сбрасываются в 0. В противном случае, если $Up = 1$, счетчик считает вверх, а когда $Up = 0$ – вниз.

Упражнение 5.52 Спроектируйте 32-разрядный счетчик, состояние которого увеличивается на 4 по каждому фронту тактового импульса. Счетчик имеет входы сброса и тактовых импульсов. После сброса все выходы счетчика устанавливаются в 0.

Упражнение 5.53 Измените счетчик из **упражнения 5.44** так, чтобы в зависимости от сигнала управления *Load* счетчик либо увеличивал свое состояние на 4, либо загружал новое 32-разрядное значение D. Когда $Load = 1$, счетчик загружает новое значение, поданное на вход D.

Упражнение 5.54 N-разрядный счетчик Джонсона (Johnson counter) состоит из N-разрядного сдвигающего регистра, имеющего вход сброса. Выход сдвигающего регистра (*Sout*) инвертируется и подается назад на его вход (*Sin*). Когда счетчик сбрасывается, все его разряды принимают нулевое значение.

a) Найдите последовательность значений на Q3:0, которая появляется на выходе 4-разрядного счетчика Джонсона непосредственно после сброса.

b) Через сколько циклов последовательность на выходе N-разрядного счетчика Джонсона будет повторяться? Объясните.

c) Спроектируйте десятичный счетчик с использованием 5-разрядного счетчика Джонсона, десяти элементов И и инверторов. Десятичный счетчик имеет входы тактового сигнала и сброса и выход Y9:0 с прямым кодированием «1 из 10». После сброса активируется выход Y0. После каждого цикла активируется следующий выход. После десяти циклов состояние счетчика повторяется. Нарисуйте схему десятичного счетчика.

d) Какие преимущества имеет счетчик Джонсона по сравнению с обычными счетчиками?

Упражнение 5.55 Создайте HDL-описание 4-разрядного сканируемого регистра, подобного приведенному на **рис. 5.37**. Промоделируйте и проведите тестирование HDL-модуля и докажите, что он работает корректно.

Упражнение 5.56 Английский язык имеет весьма большую избыточность, что позволяет восстановить искаженную передачу данных. Двоичные данные также могут быть переданы с избыточностью, которая может использоваться для исправления ошибок. Например, число 0 будет закодировано как 00000, а число 1 – как 11111. Данные передаются через зашумленный канал, который может инвертировать один или два бита. Приемник может восстановить исходные данные, если в посылке, соответствующей 0, будет, по крайней мере, три (из пяти) бита, равных 0, аналогично для 1 будет не менее трех бит, равных 1.

a) Предложите кодировку для передачи двухбитных блоков 00, 01, 10 и 11 с использованием пяти бит, которая позволяет исправлять все однобитные ошибки. *Подсказка*: кодировка 00000 и 11111 для 00 и 11, соответственно, не будет работать.
b) Спроектируйте схему, которая будет принимать пятибитный блок кодированных данных и декодировать его в двухбитный блок (00, 01, 10 и 11), даже если один бит был искажен при передаче.
c) Предположим, вы хотите использовать альтернативную пятибитовую кодировку. Как можно реализовать этот проект для обеспечения возможности изменения кодировки без замены аппаратного обеспечения?

Упражнение 5.57 Флеш EEPROM, или просто флеш-память, является относительно недавним изобретением, которое революционно изменило рынок потребительской электроники. Изучите и опишите, как работает флеш-память. Для объяснения принципа работы плавающего затвора используйте диаграммы. Опишите, как происходит запись информации в память. Оформите ссылки на использованные источники литературы.

Упражнение 5.58 Участники проекта по исследованию внеземной жизни обнаружили, что на дне озера Моно живут инопланетяне. Для классификации инопланетян по возможным планетам происхождения на основе данных NASA (зеленый или коричневый цвет кожи, слизистость, уродство) нужно создать цифровую схему. Детальные консультации с внеземными биологами привели к следующим заключениям:

- если инопланетянин 1) зеленый и слизкий или 2) уродлив, коричневый и слизкий, то он может быть марсианином;
- если существо 1) уродливое, коричневое и слизкое или 2) зеленое и неуродливое и неслизкое – оно может быть с Венеры;
- если существо 1) коричневое и неуродливое и неслизкое или 2) зеленое и слизкое – оно может быть с Юпитера.

Обратите внимание на то, что эти исследования все еще не совсем точны: например, форма жизни с пятнами зеленого и коричневого цветов, слизкая, но не уродливая, может быть с Марса или Юпитера.

a) Запрограммируйте 4×4×3 ПЛМ для идентификации пришельца. Вы можете использовать точечную нотацию.
b) Запрограммируйте 16×3 ПЗУ для идентификации пришельца. Вы можете использовать точечную нотацию.
c) Реализуйте свой проект на HDL.

Упражнение 5.59 Реализуйте следующие функции с использованием одного 16×3 ПЗУ. Для описания содержимого памяти используйте точечную нотацию.

a) $X = AB + B\overline{C}D + \overline{A}\,\overline{B}$.
b) $Y = AB + BD$.
c) $Z = A + B + C + D$.

Упражнение 5.60 Реализуйте функции из **упражнения 5.59**, с использованием 4×8×3 ПЛМ. Вы можете использовать точечную нотацию.

Упражнение 5.61 Определите размер ПЗУ, которое можно использовать для программирования следующих комбинационных схем:

a) 16-битный сумматор/вычитатель с C_{in} и C_{out};
b) умножитель 8×8;
c) 16-битный приоритетный шифратор (**упражнение 2.36**).

Является ли использование ПЗУ для реализации этих функций хорошим проектным решением? Поясните, почему да или почему нет.

Упражнение 5.62 На **рис. 5.67** приведено несколько схем, в которых используется ПЗУ. Можно ли схему в столбце I заменить схемой из столбца II той же строки при условии надлежащего программирования ПЗУ?

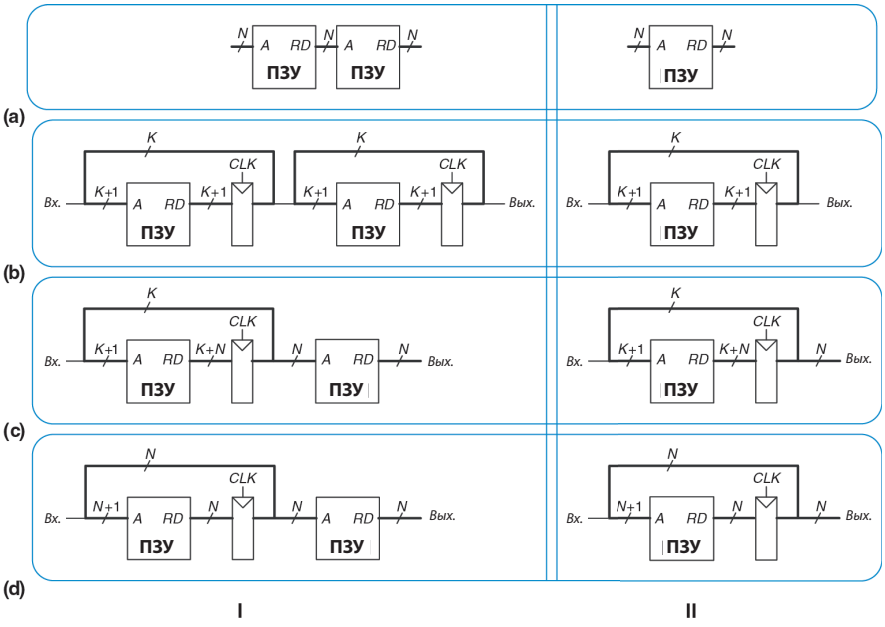


Рис. 5.67 Схемы на основе ПЗУ

Упражнение 5.63 Сколько логических элементов (LE) FPGA Cyclone IV необходимо для реализации указанных ниже функций? Покажите, как для этого нужно сконфигурировать один или несколько логических элементов. При разработке конфигурации не следует пользоваться программами синтеза:

a) комбинационная функция из **упражнения 2.13 (c)**;
b) комбинационная функция из **упражнения 2.17 (c)**;
c) функция с двумя выходами из **упражнения 2.24**;
d) функция из **упражнения 2.35**;
e) четырехвходовый приоритетный шифратор (**упражнение 2.36**).

Упражнение 5.64 Повторите **упражнение 5.63** для следующих функций:

a) восьмивходовый приоритетный шифратор (**упражнение 2.36**);
b) 3:8 дешифратор;
c) четырехразрядный сумматор с последовательным переносом (без входа и выхода переноса);

d) конечный автомат из **упражнения 3.22**;
e) счетчик, выход которого представлен в коде Грея, из **упражнения 3.27**.

Упражнение 5.65 На **рис. 5.60** приведен логический элемент FPGA Cyclone IV. В **табл. 5.7** приведены его временные параметры.

a) Какое минимальное количество логических элементов FPGA Cyclone IV необходимо для реализации показанного на **рис. 3.26** конечного автомата?
b) Чему равна максимальная тактовая частота, на которой этот конечный автомат будет стабильно работать при отсутствии расфазировки тактовых импульсов?
c) Чему равна максимальная тактовая частота, на которой этот конечный автомат будет надежно работать, если максимальная расфазировка тактовых импульсов равна 3 нс?

Упражнение 5.66 Повторите **упражнение 5.65** для конечного автомата, который показан на **рис. 3.31 (b)**.

Упражнение 5.67 Вы собираетесь использовать FPGA для реализации сортировщика леденцов. В машине будет цветовой сенсор и мотор, который отправляет красные леденцы в одну банку, а зеленые – в другую. Проект будет реализован как конечный автомат с использованием FPGA Cyclone IV. Временные характеристики FPGA приведены в **табл. 5.7**. Вы хотите, чтобы ваш конечный автомат работал на частоте 100 МГц. Какое максимальное количество логических элементов может входить в критический путь? Чему равна максимальная частота, на которой будет работать конечный автомат?

Таблица 5.7 Временные характеристики Cyclone IV

Наименование	Величина (пс)
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (одного LE)	381
t_{wire} (между LE)	246
t_{skew}	0

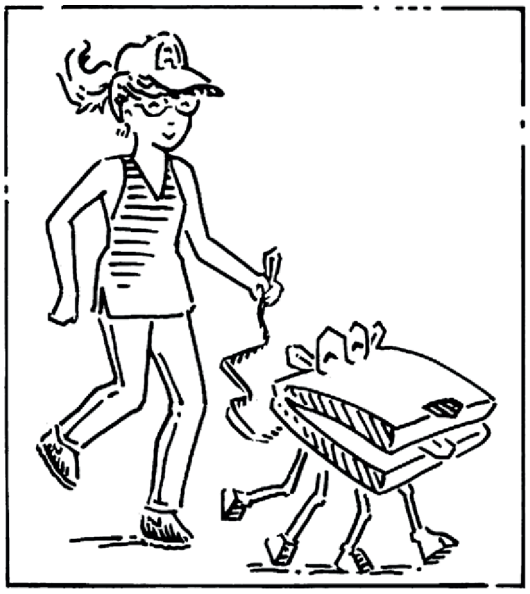
Вопросы для собеседования

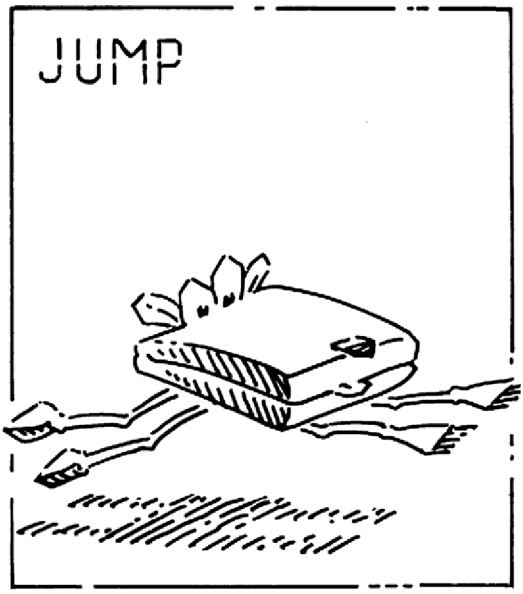
В этом разделе представлены типовые вопросы, которые могут быть заданы соискателям при поиске работы в области проектирования цифровых систем.

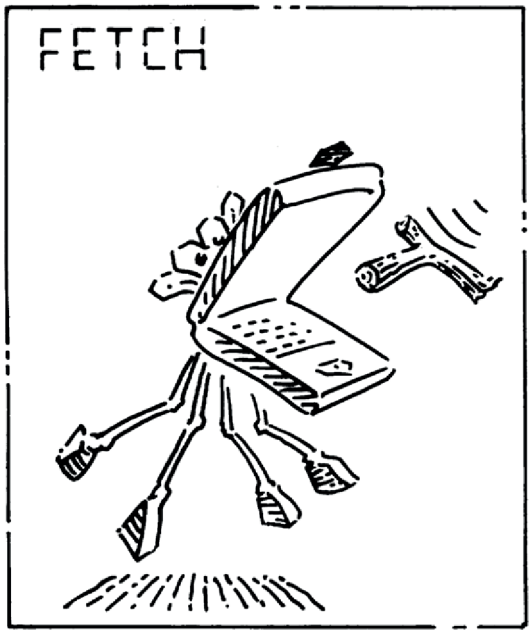
Вопрос 5.1 Чему равен наибольший возможный результат перемножения двух беззнаковых N-разрядных чисел?

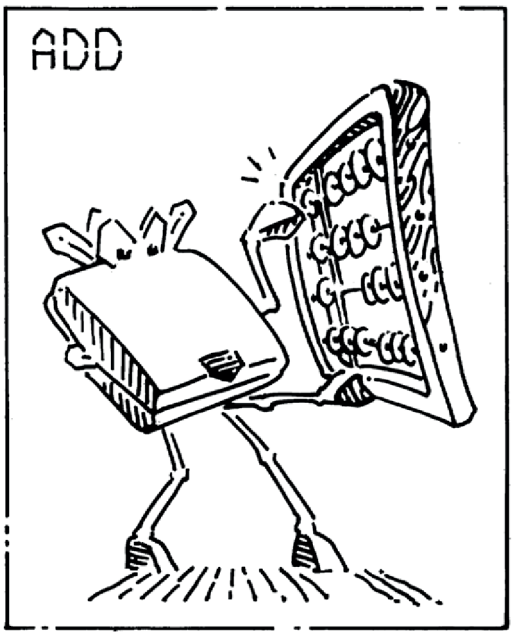
Вопрос 5.2 В *двоично-десятичном* (BCD) представлении для каждого десятичного разряда используется четыре бита. Например, 42_{10} будет представлено как 01000010_{BCD}. Объясните, почему процессор может использовать двоично-десятичное представление.

Вопрос 5.3 Разработайте сумматор, который будет складывать два беззнаковых 8-битных числа в двоично-десятичном представлении (**вопрос 5.2**). Нарисуйте схему и создайте HDL-описание вашего сумматора. Сумматор имеет входы A, B и C_{in}, выходы – S и C_{out}. Сигналы C_{in} и C_{out} представляют собой однобитный вход и выход переноса, A, B и S – 8-битные числа в двоично-десятичном представлении.



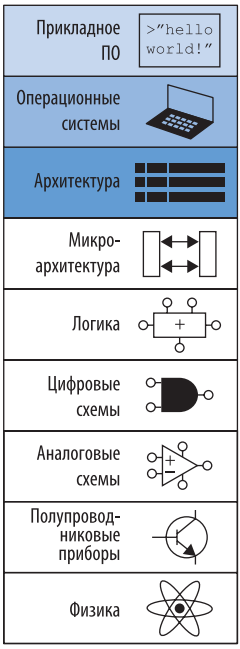

JUMP


FETCH


ADD

ГЛАВА 6

Архитектура



6.1. Предисловие

В предыдущих главах мы познакомились с принципами разработки цифровых устройств и основными цифровыми строительными блоками. В этой главе мы поднимемся на несколько уровней абстракции выше и определим *архитектуру* компьютера. Архитектура – это то, как видит компьютер программист. Она определена набором команд (языком) и местом нахождения операндов (регистры и память). Существует множество различных архитектур, таких как RISC-V, ARM, x86, MIPS, SPARC и PowerPC.

Чтобы понять архитектуру любого компьютера, нужно в первую очередь выучить его язык. Слова в языке компьютера называются «инст-

Крсте Асанович начинал создавать RISC-V как летний проект. Он работает профессором информатики в Калифорнийском университете в Беркли и занимает пост председателя правления некоммерческой организации RISC-V International, ранее известной как RISC-V Foundation. Он также является соучредителем SiFive, компании, которая разрабатывает и продает чипы, платы и дополнительные средства разработки для RISC-V.

рукциями», или «командами», а словарный запас компьютера – «системой команд»[1]. Даже сложные приложения, такие как редакторы текста и электронные таблицы, в конечном итоге состоят из последовательности таких простых команд, как сложение, вычитание и переход. Инструкция компьютера определяет операцию, которую нужно исполнить, и ее операнды. Операнды – это входные данные, с которыми производится операция, и получаемые результаты. Операнды могут находиться в памяти, в регистрах или внутри самой инструкции.

Аппаратное обеспечение компьютера «понимает» только нули и единицы, поэтому инструкции закодированы двоичными числами в формате, который называется *машинным языком*. Так же как мы используем буквы и прочие письменные символы для представления речи в виде, удобном для хранения, передачи и иных манипуляций, компьютеры используют двоичные числа, чтобы кодировать машинный язык. В архитектуре RISC-V каждая инструкция представлена 32-разрядным словом. Микропроцессоры – это цифровые системы, которые читают и выполняют команды машинного языка. Для людей чтение и разработка компьютерных программ на машинном языке представляются нудным и утомительным делом, поэтому мы предпочитаем представлять инструкции в символическом формате, который называется *языком ассемблера*.

Почти все архитектуры определяют основные инструкции, такие как сложение, вычитание и переход, которые работают с ячейками памяти или регистрами. Как только вы изучили один набор инструкций, выучить другие становится довольно просто.

Архитектура компьютера не определяет структуру аппаратного обеспечения, которое ее реализует. Зачастую существуют разные аппаратные реализации одной и той же архитектуры. Например, компании Intel и Advanced Micro Devices (AMD) производят разные микропроцессоры, которые относятся к архитектуре x86. Все они могут выполнять одни и те же программы, но при этом в их основе лежит разное аппаратное обеспечение, поэтому эти процессоры имеют разное соотношение производительности, цены и энергопотребления. Некоторые микропроцессоры оптимизированы для работы в высокопроизводительных серверах, другие оптимизированы для долгой работы батареи в ноутбуках. Взаимное

[1] Иногда говорят, что команда – это двоичное представление слов на языке компьютера, то есть представление на уровне машинных кодов, а инструкция – это понятное человеку символьное представление этих слов на любом языке, включая язык ассемблера; в этой книге мы будем считать слова «инструкция» и «команда» синонимами. – *Прим. перев.*

расположение регистров, памяти, АЛУ и других функциональных блоков, из которых состоит микропроцессор, называют *микроархитектурой*, она будет предметом **главы 7**.

В этой книге представлена архитектура RISC-V (произносится как «риск пять») – первая открытая и свободная система инструкций и процессорная архитектура с широкими перспективами коммерческого применения. Мы начнем с описания набора 32-битных целочисленных инструкций RISC-V (RV32I) версии 2.2, которые составляют ядро набора команд RISC-V, а в **разделах 6.6** и **6.7** будет рассказано про особенности других версий архитектуры. Если вы захотите углубиться в детали, то наиболее авторитетным и полным источником для вас станет «Описание набора инструкций RISC-V» (RISC-V Instruction Set Manual), доступное в интернете по адресу https://riscv.org/technical/specifications/.

Эндрю Уотерман разрабатывает микропроцессоры в SiFive, компании, которую он основал вместе с Крсте Асановичем в 2015 году, чтобы выпускать недорогие ядра RISC-V и нестандартные микросхемы. Он получил докторскую степень по информатике в Калифорнийском университете в Беркли в 2016 году, где, устав от проблем с существующими архитектурами и их неуклюжих наборов команд, он подключился к разработке RISC-V ISA и первых ядер RISC-V.

Архитектура RISC-V была впервые представлена широкой публике в 2010 году в Калифорнийском университете в Беркли ее разработчиками Крсте Асановичем, Эндрю Уотерманом, Дэвидом Паттерсоном и их единомышленниками. Архитектура RISC-V выделяется среди прочих тем, что, несмотря на открытую и бесплатную спецификацию, она сопоставима по возможностям с коммерческими архитектурами, такими как ARM и x86. Пока созданием коммерческих чипов на основе RISC-V занимаются лишь несколько компаний, включая SiFive и Western Digital, но их количество быстро растет.

Мы начнем наше погружение в архитектуру RISC-V с описания инструкций на языке ассемблера, расположения операндов и общих программных конструкций, таких как переходы, циклы, манипуляции с массивами и вызовы функций. Далее мы расскажем, как язык ассемблера переводится на машинный язык и код программы загружается в память для последующего выполнения.

В этой главе мы покажем, как архитектура RISC-V формировалась из желания разработчиков следовать четырем простым принципам, сформулированным Паттерсоном и Хеннесси:

1) для простоты придерживайтесь единообразия;
2) типичный сценарий должен быть быстрым;
3) чем меньше, тем быстрее;
4) хорошая разработка требует хороших компромиссов.

Дэвид Паттерсон работает профессором информатики в Калифорнийском университете в Беркли с 1976 года, а в 1984 году он совместно с Джоном Хеннесси изобрел вычисления с сокращенным набором инструкций. Позднее на основе этого набора была создана архитектура SPARC. Он участвовал в разработке архитектуры RISC-V и продолжает играть важную роль в ее развитии.

6.2. Язык ассемблера

Язык ассемблера – это удобное для восприятия человеком представление родного языка компьютера. Каждая инструкция языка ассемблера задает операцию, которую необходимо выполнить, а также операнды, которые будут использованы во время выполнения. Далее мы познакомим вас с простыми арифметическими инструкциями и покажем, как эти операции записываются на языке ассемблера. Затем мы определим операнды для инструкций RISC-V: регистры, ячейки памяти и константы.

В этой главе предполагается, что вы уже имеете некоторое знакомство с высокоуровневыми языками программирования, такими как C, C++ или Java (эти языки практически равнозначны для большинства примеров в данной главе, но там, где они отличаются, мы будем использовать C). В приложении C приведено введение в язык C для тех, у кого мало или совсем нет опыта программирования на этих языках.

6.2.1. Инструкции

Наиболее частая операция, выполняемая компьютером, – это сложение. В примере кода 6.1 показан код, который складывает переменные `b` и `c` и записывает результат в переменную `a`. Каждый пример сначала написан на языке высокого уровня (используется синтаксис C, C++ и Java), а затем переписан на языке ассемблера RISC-V. Не забывайте, что в языке C после команды всегда ставится точка с запятой.

Слово «мнемоника» происходит от греческого слова μμνΕσκεστης. Мнемоники языка ассемблера запомнить проще, чем наборы нулей и единиц машинного языка, представляющих ту же операцию.

Первая часть инструкции ассемблера, `add`, называется мнемоникой и определяет, какую операцию нужно выполнить. Операция осуществляется над `b` и `c`, *операндами-источниками*, а результат записывается в `a`, *операнд-назначение*[1].

Пример кода 6.1 СЛОЖЕНИЕ

Код на языке высокого уровня	Код на языке ассемблера RISC-V
`a = b + c;`	`add a, b, c`

Пример кода 6.2 демонстрирует, что вычитание похоже на сложение. Формат инструкции такой же, как у инструкции `add`, только опе-

[1] Иногда операнды-источники называют просто операндами, а операнд-назначение – результатом. – *Прим. перев.*

рация называется `sub`. Как будет показано дальше, подобное сходство есть не только у этих двух инструкций. Единообразный формат для команд является примером первого принципа хорошей разработки:

RISC-V содержит в названии слово «пять», потому что это пятая архитектура RISC, разработанная в Беркли.

Первое правило хорошей разработки:
для простоты придерживайтесь единообразия.

Пример кода 6.2 ВЫЧИТАНИЕ

Код на языке высокого уровня	Код на языке ассемблера RISC-V
`a = b - c;`	`sub a, b, c`

Инструкции с одинаковым количеством операндов – в нашем случае с двумя операндами-источниками и одним операндом-назначением (то есть с двумя операндами и одним результатом) – проще закодировать и выполнять на аппаратном уровне. Более сложный высокоуровневый код преобразуется во множество инструкций RISC-V, как показано в **примере кода 6.3**.

В примерах на языках высокого уровня однострочные комментарии начинаются с символов `//` и продолжаются до конца строки. Многострочные комментарии начинаются с `/*` и завершаются `*/`. В языке ассемблера RISC-V используются только однострочные комментарии. Они начинаются с `#` и продолжаются до конца строки. В программе на языке ассемблера в **примере кода 6.3** используется временная переменная `t` для хранения промежуточного результата операции (`b + c`).

В предисловии мы упоминали несколько симуляторов и инструментов для компиляции и моделирования ассемблерного кода C и RISC-V. В наличии также практические примеры (доступные на сайте поддержки этого учебника), в которых показано, как использовать эти инструменты.

Пример кода 6.3 БОЛЕЕ СЛОЖНЫЙ КОД

Код на языке высокого уровня	Код на языке ассемблера RISC-V

`a = b + c - d; // однострочный комментарий`
`/* многострочный`
`комментарий */`
	`add a, b, t  # a = b + t`
	`sub t, c, d  # t = c - d`

Использование нескольких инструкций ассемблера для выполнения более сложных операций является иллюстрацией второго принципа хорошей разработки компьютерной архитектуры:

Второе правило хорошей разработки:
типичный сценарий должен быть быстрым.

При использовании системы команд RISC-V типичная программа становится быстрой потому, что она включает в себя только простые

Джон Хеннесси — профессор электротехники и информатики в Стэнфордском университете; был президентом Стэнфорда с 2000 по 2016 год. Он совместно с Дэвидом Паттерсоном изобрел *вычисления с сокращенным набором инструкций*. Также разработал компьютерную архитектуру MIPS и в 1984 году стал соучредителем MIPS Computer Systems. Процессор MIPS использовался во многих коммерческих системах, включая продукты Silicon Graphics, Nintendo и Cisco. Джон Хеннесси и Дэвид Паттерсон были удостоены премии Тьюринга в 2017 году за значительный вклад в создание и развитие компьютерных архитектур.

и постоянно используемые команды. Количество команд ограничено специально, чтобы аппаратное обеспечение для их поддержки было простым и быстрым. Более сложные операции, используемые не так часто, выполняются при помощи последовательности нескольких простых команд. По этой причине RISC-V относится к компьютерным архитектурам с *сокращенным набором команд* (reduced instruction set computer, RISC). Архитектуры с большим количеством сложных инструкций, такие как архитектура x86 от Intel, называются компьютерами со *сложным набором команд* (complex instruction set computer, CISC). Например, x86 определяет инструкцию «перемещение строки», которая копирует строку (последовательность символов) из одной части памяти в другую. Такая операция требует большого количества, вплоть до нескольких сотен, простых инструкций на RISC-машине. С другой стороны, реализация сложных инструкций в архитектуре CISC требует дополнительного аппаратного обеспечения и увеличивает накладные расходы, которые замедляют выполнение простых инструкций.

Архитектура RISC использует небольшое множество различных команд, что уменьшает сложность аппаратного обеспечения и размер инструкций. Например, код операции в системе команд, состоящей из 64 простых инструкций, потребует $\log_2 64 = 6$ бит, а в системе команд из 256 сложных инструкций потребует уже $\log_2 256 = 8$ бит. В CISC-машинах сложные команды, даже если они используются очень редко, увеличивают накладные расходы на выполнение всех инструкций, включая и самые простые.

Также существуют 64- и 128-битные версии архитектуры RISC-V, но в этой книге мы будем рассматривать только 32-битный вариант. Более многоразрядные версии (RV64I и RV128I) почти идентичны 32-битной версии (RV32I), за исключением ширины регистров и адресов памяти. Основные дополнения — это инструкции, которые работают только с младшей половиной слова, и операции с памятью, которые передают более широкие слова.

6.2.2. Операнды: регистры, память и константы

Инструкции работают с операндами. В примере кода 6.1 переменные `a`, `b` и `c` являются операндами. Но компьютеры оперируют нулями и единицами, а не именами переменных. Инструкция должна знать место, откуда она сможет брать двоичные данные. Операнды могут находиться в регистрах или памяти, а еще они могут быть константами, записанными в теле самой инструкции. Компьютеры используют различные места для хранения операндов, чтобы повысить скорость исполнения и/или более эф-

фективно размещать данные. Обращение к операндам-константам или операндам, находящимся в регистрах, происходит быстро, но они могут вместить лишь небольшое количество данных. Остальные данные хранятся в емкой, но медленной памяти. Архитектуру RISC-V называют 32-битной потому, что она оперирует 32-битными данными.

Регистры

Чтобы команды могли быстро выполняться, они должны быстро получать доступ к операндам. Но чтение операндов из памяти занимает много времени, поэтому большинство архитектур предоставляют небольшое количество *регистров* для хранения наиболее часто используемых операндов. Архитектура RISC-V использует 32 регистра, которые называют *набором регистров*, или *регистровым файлом*. Чем меньше количество регистров, тем быстрее к ним доступ. Это приводит нас к третьему правилу хорошей разработки компьютерной архитектуры:

Третье правило хорошей разработки:
чем меньше, тем быстрее.

Найти необходимую информацию получится гораздо быстрее в небольшом количестве тематически подобранных книг, лежащих на столе, а не в многочисленных книгах, находящихся на полках в библиотеке. То же самое и с чтением данных из регистров и памяти. Прочитать данные из небольшого набора регистров (например, из 32 регистров) можно гораздо быстрее, чем из 1000 регистров или из большой памяти. Небольшие регистровые файлы обычно состоят из маленького массива памяти SRAM (**раздел 5.5.3**).

В **примере кода 6.4** показана инструкция `add` с регистровыми операндами. Переменные `a`, `b` и `c` произвольно размещены в регистрах `s0`, `s1` и `s2`. Имя `s1` произносят как «регистр s1» или просто «s1». Инструкция складывает 32-битные значения, хранящиеся в `s1` (`b`) и `s2` (`c`), и записывает 32-битный результат в `s0` (`a`). **Пример кода 6.5** демонстрирует разработанный на ассемблере RISC-V код, использующий временный регистр `t0` для вычисленного промежуточного значения `c - d`.

В приложении B, которое находится в конце учебника, представлен удобный обзор полного набора инструкций RISC-V.

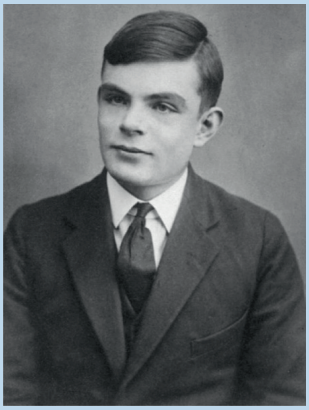
Пример кода 6.4 РЕГИСТРОВЫЕ ОПЕРАНДЫ

Код на языке высокого уровня	Код на языке ассемблера RISC-V
`a = b + c;`	`# s0 = a, s1 = b, s2 = c` `add s0, s1, s2  # a = b + c`

Пример кода 6.5 РЕГИСТРОВЫЕ ОПЕРАНДЫ

Код на языке высокого уровня	Код на языке ассемблера RISC-V

`a = b + c - d;`	`# s0 = a, s1 = b, s2 = c, s3 = d, t0 = t`
	`add t0, s1, s2  # t = b + c`
	`sub s0, t0, s3  # a = t - d`



Алан Тьюринг, 1912–1954 гг.

Британский математик и ученый по компьютерным наукам, который считается основоположником теоретической информатики и искусственного интеллекта. Он прославился как изобретатель машины Тьюринга — математической модели вычислений, представляющей абстрактный процессор. Он также разработал электромеханическую машину для расшифровки зашифрованных сообщений во время Второй мировой войны, что приблизило окончание войны и спасло миллионы жизней. Премия Тьюринга, которая является высшей наградой в области вычислительной техники, была названа в его честь и вручается ежегодно с 1966 года. В настоящее время она включает сопутствующий денежный приз в размере 1 млн долларов.

Пример 6.1 ТРАНСЛЯЦИЯ КОДА ИЗ ЯЗЫКА ВЫСОКОГО УРОВНЯ В ЯЗЫК АССЕМБЛЕРА

Преобразуйте приведенный ниже код, написанный на языке высокого уровня, в код на языке ассемблера[1]. Считайте, что переменные `a`, `b` и `c` находятся в регистрах `s0`–`s2`, а `f`, `g`, `h`, `i` и `j` — в регистрах `s3`–`s7`.

```
// код на языке высокого уровня
a = b - c;
f = (g + h) - (i + j);
```

Решение Программа использует четыре ассемблерные инструкции.

```
# Код на языке ассемблера RISC-V
# s0 = a, s1 = b, s2 = c, s3 = f, s4 = g, s5 = h, s6 = i,
s7 = j
  sub s0, s1, s2      # a = b - c
  add t0, s4, s5      # t0 = g + h
  add t1, s6, s7      # t1 = i + j
  sub s3, t0, t1      # f = (g + h) - (i + j)
```

Набор регистров

В **табл. 6.1** перечислены имена, порядковые номера и назначение каждого из 32 регистров RISC-V. У каждого регистра есть номер от 0 до 31 и специальное имя для обозначения обычного назначения регистра. Для лучшей читаемости кода инструкции ассемблера обычно используют специальные имена, например `s1`, но они также могут использовать номер регистра (например, `x9` для регистра номер 9). В нулевом регистре всегда хранится константа 0; попытка записать в него другое значение игнорируется. Регистры от `s0` до `s11` (регистры 8–9 и 18–27) и от `t0` до `t6` (регистры 5–7 и 28–31) используются для хранения переменных; `ra` и регистры от `a0` до `a7` служат для

[1] Трансляцией называется процесс преобразования программы, написанной на одном языке программирования, в программу на другом языке. — *Прим. перев.*

вызовов функций, как описано в **разделе 6.3.7**. Регистры 2–4 носят имена `sp`, `gp` и `tp`. Они будут описаны позже.

Таблица 6.1 Набор регистров RISC-V

Название	Номер	Назначение
zero	x0	Константа нуля
ra	xl	Адрес возврата (от англ. *return address*)
sp	x2	Указатель стека (от англ. *stack pointer*)
gp	x3	Глобальный указатель (от англ. *global pointer*)
Tp	x4	Указатель потока (от англ. *thread pointer*)
t0–t2	x5–x7	Временные переменные
s0/fp	x8	Сохраняемая переменная / Указатель фрейма стека
s1	x9	Сохраняемая переменная
a0–a1	x10–x11	Аргументы функций / Возвращаемые значения
a2–a7	x12–x17	Аргументы функций
s2–s11	x18–x27	Сохраняемые переменные
t3–t6	x28–x31	Временные переменные

Константы / непосредственные операнды

Помимо операций с регистрами, инструкции RISC-V могут использовать *константы*, или *непосредственные операнды* (immediate). Они получили такое название, потому что их значения доступны непосредственно из команды и не требуют обращения к регистру или памяти. **Пример кода 6.6** демонстрирует инструкцию `addi` (add immediate), которая прибавляет константу к значению регистра. В ассемблерном коде непосредственный операнд может быть записан в десятичном, шестнадцатеричном или двоичном формате. Шестнадцатеричные константы в языке ассемблера RISC-V начинаются с `0x`, а двоичные начинаются с `0b`, как и в C. Непосредственные операнды представляют собой 12-битные числа в дополнительном коде, поэтому они дополняются знаковым битом до 32 бит. Инструкция `addi` – удобный способ инициализировать значения регистров небольшими константами. **Пример кода 6.7** демонстрирует инициализацию переменных `i`, `x` и `y` значениями 0, 2032 и –78 соответственно.

Непосредственные операнды могут быть записаны в десятичном, шестнадцатеричном или двоичном формате. Например, все следующие инструкции записывают десятичное значение 109 в регистр `s5`:

```
addi s5, x0,0b1101101
addi s5, x0,0x6D
addi s5, x0,109
```

Пример кода 6.6 НЕПОСРЕДСТВЕННЫЕ ОПЕРАНДЫ

Код на языке высокого уровня	Код на языке ассемблера RISC-V

`a = a + 4;`
`b = a − 12;`
	`# s0 = a, s1 = b`
	`addi s0, s0, 4 # a = a + 4`
	`addi s1, s0, −12 # b = a − 12`

Пример кода 6.7 ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ

Код на языке высокого уровня	Код на языке ассемблера RISC-V

`i = 0;`
`x = 2032;`
`y = −78;`
	`# s4 = i, s5 = x, s6 = y`
	`addi s4, zero, 0 # i = 0`
	`addi s5, zero, 2032 # x = 2032`
	`addi s6, zero, −78 # y = −78`

Чтобы использовать константы большего размера, используйте инструкцию непосредственной записи в старшие разряды `lui` (load upper immediate), за которой следует инструкция непосредственного сложения `addi`, как показано в **примере кода 6.8**. Инструкция `lui` загружает 20-битное значение сразу в 20 старших битах и помещает нули в младшие биты.

Пример кода 6.8 ЗАПИСЬ 32-БИТНОЙ КОНСТАНТЫ В РЕГИСТР

Код на языке высокого уровня	Код на языке ассемблера RISC-V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE # s2 = 0xABCDE000` `addi s2, s2, 0x123 # s2 = 0xABCDE123`

Тип данных `int` в C представляет число со знаком, то есть целое число в дополнительном коде. Спецификация C требует, чтобы число типа `int` имело разрядность *не менее* 16 бит, но не указывает определенный размер. Большинство современных компиляторов (в том числе для RV32I) используют 32 бита, поэтому `int` представляет число в диапазоне $[-2^{31}, 2^{31}-1]$. C также определяет `int32_t` как 32-битное целое число в дополнительном коде, но эта запись длиннее.

При использовании многоразрядных непосредственных операндов, если указанный в `addi` 12-битный непосредственный операнд отрицательный (т. е. бит 11 равен 1), старшая часть постоянного значения в `lui` должна быть увеличена на единицу. Помните, что *знак* `addi` расширяет 12-битное непосредственное значение, поэтому отрицательное непосредственное значение будет содержать все единицы в своих старших 20 битах. Поскольку в дополнительном коде все единицы означают число −1, добавление числа, у которого все разряды установлены в 1, к старшим разрядам непосредственного операнда приводит к вычитанию 1 из этого числа. **Пример кода 6.9** иллюстрирует ситуацию, когда мы хотим получить постоянное значение `0xFEEDA987`. Инструкция `lui s2, 0xFEEDB` записывает `0xFEEDB000` в регистр `s2`. Как видите, число, которое нужно записать в старшие 20 разрядов (`0xFEEDA`),

предварительно увеличено на 1. `0x987` – это 12-битное представление числа `-1657`, поэтому инструкция `addi s2, s2, -1657` выполняет сложение `s2` и непосредственного 12-битного числа в дополнительном коде (`0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987`) и помещает нужный нам результат в `s2`.

Пример кода 6.9 32-БИТНАЯ КОНСТАНТА С 1 В РАЗРЯДЕ 11

Код на языке высокого уровня

```
int a = 0xFEEDA987;
```

Код на языке ассемблера RISC-V

```
lui s2, 0xFEEDB     # s2 = 0xFEEDB000
addi s2, s2, -1657  # s2 = 0xFEEDA987
```

Память

Если бы операнды хранились только в регистрах, то мы могли бы разрабатывать лишь простые программы, содержащие не более 32 переменных. Поэтому данные также можно хранить в памяти. По сравнению с регистровым файлом, память имеет много места для хранения данных, но доступ к ней занимает больше времени. По этой причине часто используемые переменные хранятся в регистрах. Комбинируя память и регистры, программа может получать доступ к большим объемам данных достаточно быстро. Как было описано в **разделе 5.5**, память устроена как массив слов с данными. Архитектура RISC-V RV32I использует 32-битные адреса памяти и 32-битные слова с данными.

RISC-V применяет память с *побайтовой адресацией*. Это значит, что каждый байт памяти имеет уникальный адрес, как показано на **рис. 6.1 (a)**. Поскольку 32-битное слово состоит из четырех 8-битных байтов, то адрес каждого слова (word address) кратен 4.

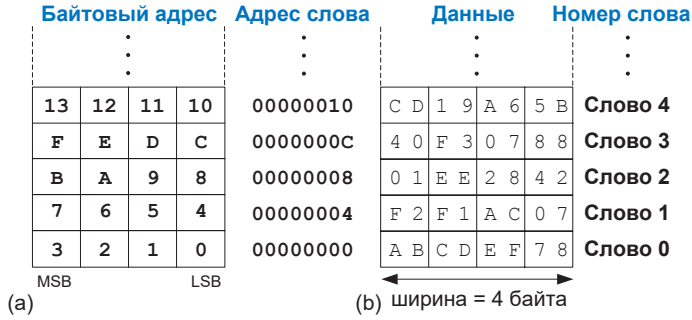


Рис. 6.1 Память RISC-V с побайтовой адресацией: адрес байта (a) и данные (b)

Старший байт (most significant byte,MSB) находится слева, а *младший байт* (least significant byte, LSB) – справа. Порядок байтов в слове мы обсудим немного позже в **разделе 6.6.1**. И 32-битный адрес слова,

и значение данных на **рис. 6.1 (b)** даны в шестнадцатеричном формате. Например, слово данных 0xF2F1AC07 хранится по адресу в памяти с номером 4. По общепринятому соглашению в схематичном виде память изображают так, чтобы младшие адреса памяти находились внизу, а старшие – вверху.

Многие версии RISC-V требуют, чтобы для инструкций lw и sw применялись только *адреса с выравниванием по словам*, т. е. адреса слов, которые делятся на четыре. Одни архитектуры, такие как x86, допускают чтение и запись данных без выравнивания по словам, но другие в целях упрощения нуждаются в строгом выравнивании. В этом учебнике мы предполагаем строгое выравнивание. Конечно, адреса в байтах для инструкций загрузки и сохранения байтов lb и sb (раздел 6.3.6) не нужно выравнивать по словам.

Инструкция загрузки слова `lw` (load word) считывает слово данных из памяти в регистр. В **примере кода 6.10** демонстрируется загрузка слова в памяти под номером 2, расположенного по адресу 8, в `a(s7)`. В языке C число в скобках – это *индекс* или номер слова, которые мы обсудим далее в **разделе 6.3.6**. Инструкция `lw` задает адрес памяти, используя *смещение*, добавленное к *базовому регистру*. Напомним, что каждое слово данных состоит из 4 байтов, поэтому адрес слова в четыре раза больше номера слова. Слово номер 0 находится по адресу 0, слово 1 – по адресу 4, слово 2 – по адресу 8 и т. д. В этом примере к базовому регистру (ноль) добавляется смещение 8, и получается адрес 8 или слово 2. После выполнения инструкции загрузки слова `lw` в **примере кода 6.10** регистр `s7` содержит значение 0x01EE2842, которое представляет собой данные, извлеченные из ячейки памяти 8 на **рис. 6.1**.

Пример кода 6.10 ЧТЕНИЕ ПАМЯТИ

Код на языке высокого уровня

```
a = mem[2];
```

Код на языке ассемблера RISC-V

```
# s7 = a
lw s7, 8(zero) # s7 = данные по адресу памяти (zero + 8)
```

Инструкция сохранения слова `sw` переносит слово данных из регистра в память. **Пример кода 6.11** демонстрирует запись значения 42 из регистра `t3` в слово памяти 5, расположенное по адресу 20.

Пример кода 6.11 ЗАПИСЬ В ПАМЯТЬ

Код на языке высокого уровня

```
mem[5] = 42;
```

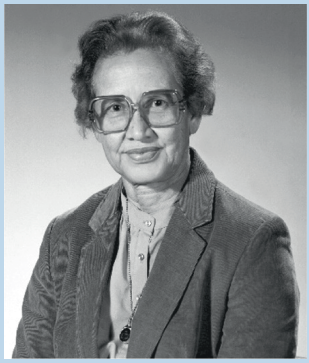
Код на языке ассемблера RISC-V

```
addi t3, zero, 42    # t3 = 42
sw t3, 20(zero)      # данные по адресу 20 = 42
```

6.3. Программирование

Языки программирования, подобные C и Java, называют *языками программирования высокого уровня* потому, что они предоставляют программисту возможность разрабатывать программы, используя абстрак-

ции более высокого уровня, чем те, что имеются в языке ассемблера. Большинство языков программирования высокого уровня используют весьма общие программные конструкции, такие как арифметические и логические операции, операторы `if/else`, циклы `for` и `while`, индексирование массивов и вызовы функций. В приложении C приведено больше примеров таких конструкций из языка C. В этом разделе мы узнаем, как можно реализовать такие высокоуровневые конструкции на ассемблере RISC-V.



Кэтрин Джонсон, 1918–2020 гг.

Креола Кэтрин Джонсон была математиком, компьютерным ученым, а заодно и одной из первых афроамериканок, работавших в НАСА. В 18 лет она окончила университет Западной Вирджинии с отличием со степенью бакалавра математики и французского языка. Когда она пришла в НАСА, то первое время работала «компьютером» в буквальном смысле этого слова — входила в состав особой группы вычислителей, в основном женщин, которые вручную выполняли точные расчеты. В 1961 году Джонсон рассчитала траекторию полета Алана Шепарда, первого американского космонавта. До этого никогда в истории НАСА имена женщин-исполнителей не указывали в отчетах, даже когда они выполняли большую часть работы. Коллеги из НАСА доверяли ее расчетам, поэтому для Джонсон было легче продвигать внедрение компьютеров для автоматизации вычислений. Президент Барак Обама наградил ее Президентской медалью свободы в 2015 году.

6.3.1. Порядок выполнения программы

Как и данные, команды хранятся в памяти. Каждая команда имеет длину 32 бита (4 байта), поэтому последовательные адреса инструкций увеличиваются на четыре. Например, в приведенном ниже фрагменте кода инструкция `addi` находится в памяти по адресу 0x538, а следующая инструкция `lw` находится по адресу 0x53C.

Адрес памяти	Инструкция
0x538	`addi s1, s2, s3`
0x53C	`lw t2, 8(s1)`
0x540	`sw s3, 3(t6)`

Адрес текущей команды хранится в специальном регистре, который называют *счетчиком команд* (program counter, PC). Его значение увеличивается на четыре после завершения каждой инструкции, чтобы процессор мог извлечь следующую инструкцию из памяти. Например, когда выполняется инструкция `addi`, счетчик команд хранит значение 0x538. После завершения операции сложения содержимое счетчика команд увеличивается на 4 (0x53C), и процессор извлекает расположенную по этому адресу инструкцию `lw`.

6.3.2. Арифметические/логические инструкции

В архитектуре RISC-V определены разнообразные арифметические и логические инструкции. Сейчас мы кратко с ними ознакомимся, поскольку они пригодятся нам в дальнейшем для построения высокоуровневых программных конструкций.

Логические инструкции

В архитектуре RISC-V имеются *логические операции* `and`, `or` и `xor`. Соответствующие им одноименные инструкции производят побитовые операции над значениями двух регистров-источников и помещают результат в регистр-назначение, как показано на **рис. 6.2**. Версии этих логических инструкций с непосредственным операндом – `andi`, `ori` и `xori` – используют один регистр-источник и 12-битный непосредственный операнд, дополненный битом знака[1].

	Регистры-источники			
s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Код ассемблера		Результат			
and s3, s1, s2	s3	0100 0110	1010 0001	0000 0000	0000 0000
or s4, s1, s2	s4	1111 1111	1111 1111	1111 0001	1011 0111
xor s5, s1, s2	s5	1011 1001	0101 1110	1111 0001	1011 0111

Рис. 6.2 Логические операции

Инструкция `and` полезна для *наложения маски* (masking) на биты, т. е. для обнуления ненужных битов. Например, инструкция `and` на **рис. 6.2** обнуляет биты в `s1` в соответствии с нулевыми битами в `s2`. В данном случае обнуляются два младших байта `s1`. Два старших незамаскированных байта `s1` (0x46A1) помещаются в `s3`. Маска может быть наложена на любое подмножество битов регистра. Например, чтобы обнулить бит 3 `s0` и поместить результат в `s6`, воспользуйтесь инструкцией `andi s6, s0, 0xFF7`.

Инструкцию `or` хорошо использовать для объединения битов из двух регистров. Например, в результате операции `0x347A0000 OR 0x000072FC = 0x347A72FC` мы получим комбинацию двух значений. Эту инструкцию также можно использовать для *установки* битов в регистре (т. е. для присвоения им значения 1). Например, инструкция `ori s7, s0, 0x020` устанавливает бит 5 регистра `s0` в единицу и помещает результат в `s7`.

В архитектуре RISC-V отсутствует операция `not`, но ее можно выполнить с помощью инструкции `xori s8, s1, –1`. Напомним, что значение 1 (0xFFF) расширяется знаковым битом до 0xFFFFFFFF (все единицы). Логическая операция XOR со всеми единицами инвертирует все биты, поэтому в `s8` сохранится дополнение `s1` до единицы (обратный код).

[1] Дополнение непосредственных операндов логических операций знаковым битом выглядит довольно необычно. Многие другие архитектуры, такие как MIPS и ARM, дополняют такие операнды нулем.

Инструкции сдвига

Базовый набор инструкций RISC-V в настоящее время не содержит какие-либо команды побитовой обработки, кроме сдвигов. Некоторые варианты набора инструкций также содержат команды циклического сдвига, а еще выборочного сброса и установки отдельных битов и т. д. С 2021 г. планировалось добавить стандартное расширение «B» RISC-V для побитовых операций, но на момент подготовки этого учебника оно не было завершено.

Инструкции сдвига сдвигают значение в регистре влево или вправо, отбрасывая биты с конца. Архитектура RISC-V поддерживает следующие операции сдвига: `sll` (логический сдвиг влево, shift left logical), `srl` (логический сдвиг вправо, shift right logical) и `sra` (арифметический сдвиг вправо, shift right arithmetic). Как уже обсуждалось в **разделе 5.2.5**, при сдвиге влево освобождающиеся младшие биты всегда заполняются нулями. Но сдвиг вправо может быть как логическим (старшие значащие биты заполняются нулями), так и арифметическим (старшие значащие биты заполняются значением знакового бита). Величина сдвига определяется значением во втором регистре-источнике. Также доступны версии каждой инструкции с непосредственным операндом (`slli`, `srli` и `srai`), где величину сдвига определяет 5-битное беззнаковое непосредственное значение.

На **рис. 6.3** показан ассемблерный код и значения регистров после выполнения инструкций `slli`, `srli` и `srai` с непосредственным операндом. Значение в регистре s5 сдвигается на указанную величину, а результат помещается в регистр-назначение.

		Регистры-источники			
	s5	1111 1111	0001 1100	0001 0000	1110 0111

Код ассемблера		Результат			
`slli t0, s5, 7`	**t0**	1000 1110	0000 1000	0111 0011	1000 0000
`srli s1, s5, 17`	**s1**	0000 0000	0000 0000	0111 1111	1000 1110
`srai t2, s5, 3`	**t2**	1111 1111	1110 0011	1000 0010	0001 1100

Рис. 6.3 Инструкции сдвига с непосредственными операндами

Как обсуждалось в **разделе 5.2.5**, сдвиг значения влево на N бит эквивалентен его умножению на 2^N. Например, `slli s0, s0, 3` умножает `s0` на 8 (т. е. 2^3). Аналогично сдвиг значения вправо на N бит эквивалентен его делению на 2^N. Арифметический сдвиг вправо делит числа в дополнительном коде, а логический сдвиг вправо делит числа без знака.

Логические сдвиги также можно использовать совместно с инструкциями `and` и `or` для извлечения или формирования *битовых полей*. Например, следующий код извлекает биты с 15 по 8 из `s7` и помещает их в младший байт `s6`. Если `s7` содержит значение 0x1234ABCD, то после завершения этого кода в `s6` окажется значение 0xAB.

```
srli s6, s7, 8
andi s6, s6, 0xFF
```

Инструкции умножения

Умножение несколько отличается от других арифметических операций, потому что умножение двух N-битных чисел дает 2^N-битное произведение. Архитектура RISC-V содержит разные варианты инструкции умножения, которые дают 32- или 64-битные произведения. Эти инструкции не являются частью набора RV32I, но включены в расширение RVM (RISC-V multiply/divide, умножение/деление RISC-V).

Инструкция *умножения* `mul` (multiply) перемножает два 32-битных числа и возвращает 32-битное произведение. Например, инструкция `mul s1, s2, s3` перемножает значения в `s2` и `s3` и помещает младшие 32 бита произведения в `s1`; самые старшие значащие 32 бита произведения отбрасываются. Эта инструкция полезна для умножения небольших чисел, когда их произведение гарантированно умещается в 32 бита. Младшие 32 бита произведения не зависят от того, рассматриваем ли мы знак операндов.

Существуют три варианта операции «умножения старших разрядов»: `mulh`, `mulhsu` и `mulhu`. Эти инструкции помещают в регистр назначения старшие 32 бита произведения. Инструкция `mulh` (multiply high signed signed, умножение старших разрядов с учетом знаков) рассматривает оба операнда как числа со знаком. Инструкция `mulhsu` (multiply high signed unsigned, умножение старших разрядов с одним знаком) рассматривает первый операнд как число со знаком, а второй – как число без знака, а `mulhu` (multiply high unsigned unsigned, умножение старших разрядов без знаков) обрабатывает оба операнда как беззнаковые. Например, `mulhsu t1, t2, t3` рассматривает `t2` как 32-битное число со знаком (в дополнительном коде), а `t3` как 32-битное число без знака, перемножает два этих исходных операнда и помещает старшие 32 бита результата в `t1`. Чтобы поместить весь 64-битный результат 32-битного умножения в два регистра, назначенных пользователем, нужно последовательно воспользоваться двумя инструкциями – сначала инструкцией «умножения старших разрядов», а затем инструкцией `mul`. Например, следующий код умножает 32-битные числа со знаком в `s3` и `s5` и помещает 64-битное произведение в `t1` и `t2`. Можно сказать, что {`t1`, `t2`} = `s3` × `s5`.

```
mulh t1, s3, s5
mul t2, s3, s5
```

6.3.3. Ветвление программ

Программы были бы скучными и не очень полезными, если бы они могли выполняться каждый раз только в одном и том же порядке, независимо от входных данных. Преимуществом компьютера над калькулятором является способность принимать решения. Компьютер выполняет разные задачи в зависимости от входных данных. Например, операторы `if/`

`else`, операторы `switch/case`, циклы `while` и `for` выполняют те или иные части кода в зависимости от результата проверки какого-либо условия. Инструкции переходов изменяют счетчик программы, для того чтобы пропустить некоторые участки кода или повторить предыдущий код. Инструкции *условных переходов*, также называемые инструкциями *ветвления* (branch), проверяют какое-либо условие и осуществляют переход только в том случае, если проверка возвращает ИСТИНУ. Инструкции *безусловного перехода* (jump) осуществляют переход всегда.

Условные переходы

Система команд RISC-V содержит шесть инструкций условного перехода, каждая из которых принимает два регистра-источника и метку, указывающую на место кода, куда осуществляется переход. Инструкция `beq` (переход при равенстве) срабатывает, когда значения в двух регистрах-источниках равны. Инструкция `bne` (переход при неравенстве) срабатывает, когда регистры-источники не совпадают. Переход по инструкции `blt` (переход, если меньше) происходит, когда значение в первом регистре-источнике меньше, чем значение во втором, а `bge` (переход, если больше или равно) срабатывает, когда первое значение больше или равно второму. Инструкции `blt` и `bge` обрабатывают операнды как числа со знаком, а `bltu` и `bgeu` обрабатывают операнды как беззнаковые.

Нет необходимости в инструкциях `bgt` или `ble`, потому что нужный результат можно получить путем перестановки местами исходных регистров `blt` и `bge`. Тем не менее они доступны в виде псевдоинструкций (раздел 6.3.8).

Пример кода 6.12 иллюстрирует использование инструкции `beq`. Когда программа доходит до этой инструкции, значение в `s0` равно значению в `s1`, поэтому осуществляется переход, и следующей выполненной инструкцией будет инструкция `add`, расположенная сразу после метки с именем `target`. Инструкции `addi` и `sub`, расположенные между инструкцией ветвления и меткой, не выполняются.

Пример кода 6.12 УСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `beq`

Код на языке ассемблера RISC-V

```
 addi s0, zero, 4      # s0 = 0 + 4 = 4
 addi s1, zero, 1      # s1 = 0 + 1 = 1
 slli s1, s1, 2        # s1 = 1 << 2 = 4
 beq s0,  s1, target   # s0 = = s1, переход происходит
 addi     s1, s1, 1    # не выполняется
 sub      s1, s1, s0   # не выполняется
target:               # метка
 add      s1, s1, s0   # s1 = 4 + 4 = 8
```

Метки в ассемблерном коде являются ссылками на инструкции программы. Когда ассемблерный код транслируется в машинный, метки заменяются соответствующими адресами инструкций (**разделы 6.4.3**

и **6.4.4**). Определяя новую метку непосредственно перед инструкцией, на которую она будет ссылаться, мы ставим двоеточие после имени метки. Большинство программистов делают отступы из пробелов или символов табуляции перед инструкциями, но не делают их перед метками, что позволяет визуально выделить метки среди остального кода.

Пример кода 6.13 демонстрирует использование инструкции перехода при неравенстве (`bne`). В этом случае переход не осуществляется потому, что `s0` равен `s1`, и процессор продолжает выполнять код, расположенный сразу после инструкции `bne`. В этом фрагменте кода выполняются все инструкции.

Пример кода 6.13 УСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `bne`

Код на языке ассемблера RISC-V

```
 addi s0, zero, 4     # s0 = 0 + 4 = 4
 addi s1, zero, 1     # s1 = 0 + 1 = 1
 slli s1, s1, 2       # s1 = 1 << 2 = 4
 bne  s0, s1, target  # переход не происходит
 addi s1, s1, 1       # s1 = 4 + 1 = 5
 sub  s1, s1, s0      # s1 = 5 – 4 = 1
target:
 add  s1, s1, s0      # s1 = 1 + 4 = 5
```

Безусловные переходы

Для безусловных переходов программа может использовать инструкции трех типов: обычный *безусловный переход* `j` (jump), *безусловный переход с возвратом* `jal` (jump and link) и *безусловный переход по регистру* `jr` (jump register). Безусловный переход (`j`) осуществляет переход к инструкции, следующей за указанной меткой. **Пример кода 6.14** иллюстрирует использование инструкции `j` , после которой программа пропустит следующие три инструкции и продолжит выполнение с инструкции `add`, расположенной после метки `target`. Оставшиеся инструкции `jal` и `jr` мы подробно обсудим в **разделе 6.3.7**, где они используются для вызовов функций.

Пример кода 6.14 БЕЗУСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `j`

Код на языке ассемблера RISC-V

```
 j target         # переход к метке target
 srai s1, s1, 2   # не выполняется
 addi s1, s1, 1   # не выполняется
 sub s1, s1, s0   # не выполняется
target:
 add s1, s1, s0   # s1 = s1 + s0
```

6.3.4. Условные операторы

Операторы `if`, `if/else` и `switch/case` являются условными операторами, которые часто используются в языках высокого уровня. Каждый из этих операторов при выполнении определенного условия выполняет участок кода, состоящий, в свою очередь, из одного или нескольких операторов. В этом разделе показано, как перевести эти высокоуровневые конструкции на язык ассемблера RISC-V.

Оператор if

Оператор `if` выполняет участок кода, называемый *блоком «если»* (if block), только если выполняется заданное условие. **Пример кода 6.15** демонстрирует, как перевести выражение с оператором `if` на язык ассемблера RISC-V. Код на языке ассемблера для оператора if проверяет условие, противоположное условию, заданному на языке высокого уровня. В **примере кода 6.15** код на языке высокого уровня проверяет условие `яблоки == апельсины`, а ассемблерный код проверяет условие `яблоки != апельсины`. Инструкция `bne` осуществляет переход, пропуская блок *«если»*, когда условие не выполняется. В противном случае (т. е. когда `яблоки == апельсины`) переход не происходит и выполняется блок *«если»*.

В языке C и многих других языках программирования высокого уровня двойной знак равенства == является проверкой на равенство, возвращающей ИСТИНУ, если значения по обе стороны двойного равенства совпадают. Запись != означает проверку на неравенство.

Пример кода 6.15 ОПЕРАТОР `if`

Код на языке высокого уровня

```
if (apples = = oranges)
  f = g + h;
apples = oranges – h;
```

Код на языке ассемблера RISC-V

```
# s0 = яблоки, s1 = апельсины
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # переход, если (apples != oranges)
    add s2, s3, s4 # f = g + h
L1: sub s0, s1, s4 # апельсины = яблоки – h
```

Операторы if/else

Операторы `if/else` выполняют один из двух участков кода в зависимости от условия. Когда выполнено условие выражения `if`, выполняется блок *«если»*. В противном случае выполняется блок *«иначе»* (else block). **Пример кода 6.16** демонстрирует пример оператора `if/else`.

Как и в случае оператора if, ассемблерный код для оператора if/else проверяет условие, противоположное условию, заданному в коде на языке высокого уровня. Так, в **примере кода 6.16** код высокого уровня проверяет условие `(яблоки == апельсины)`, а ассемблерный код проверяет условие `(яблоки != апельсины)`. Если это противоположное

условие истинно, то инструкция `bne` пропускает блок *«если»* и выполняет блок *«иначе»*. В противном случае блок *«если»* выполняется и завершается инструкцией безусловного перехода `j` для перехода на участок после блока *«иначе»*.

Пример кода 6.16 ОПЕРАТОР `if/else`

Код на языке высокого уровня

```
if (apples = = oranges)
  f = g + h;
else
  apples = oranges – h;
```

Код на языке ассемблера RISC-V

```
# s0 = яблоки, s1 = апельсины
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # пропуск, если (яблоки != апельсины)
    add s2, s3, s4 # f = g + h
    j L2
L1: sub s0, s1, s4 # яблоки = апельсины – h
L2:
```

Операторы switch/case

Операторы `switch/case`, также называемые просто операторами `case`, выполняют один из нескольких участков кода в зависимости от того, какое из данных условий выполняется. Если ни одно из условий не выполнено, то выполняется блок `default`. Оператор `case` аналогичен последовательности вложенных операторов `if/else`. **Пример кода 6.17** демонстрирует два фрагмента на языке высокого уровня с одной и той же функциональностью: они вычисляют, какую купюру следует выдать в банкомате (automatic teller machine, ATM) – 20, 50 или 100 долларов – в зависимости от нажатой кнопки. Реализация на языке ассемблера RISC-V одинакова для обоих фрагментов кода высокого уровня. Номер нажатой кнопки хранится в переменной `button`, номинал купюры, которую нужно выдать, сохраняется в переменной `amt`.

6.3.5. Циклы

Циклы многократно выполняют участок кода в зависимости от условия. Операторы `for` и `while` являются обычными конструкциями для организации циклов в языках высокого уровня. В этом разделе будет показано, как, используя условный переход, реализовать их на языке ассемблера RISC-V.

Цикл while

Цикл `while` многократно выполняет участок кода до тех пор, пока условие не станет ложным. В **примере кода 6.18** цикл `while` ищет значение `x` такое, чтобы $2^x = 128$. Цикл выполнится семь раз, прежде чем достигнет условия `pow` = 128.

Пример кода 6.17 ОПЕРАТОРЫ `switch/case`

Код на языке высокого уровня

```
switch (button) {
case 1: amt = 20; break;
case 2: amt = 50; break;
case 3: amt = 100; break;
default: amt = 0;
}

// аналогичный код
с использованием
// операторов if/else
if (button = = 1) amt = 20;
else if (button = = 2) amt =
50;
else if (button = = 3) amt =
100;
else amt = 0;
```

Код на языке ассемблера RISC-V

```
# s0 = button, s1 = amt

case1:
  addi t0, zero, 1    # t0 = 1
  bne s0, t0, case2   # button = = 1?
  addi s1, zero, 20   # если да, amt = 20
  j done              # выход из блока case
case2:
  addi t0, zero, 2    # t0 = 2
  bne s0, t0, case3   # button = = 2?
  addi s1, zero, 50   # если да, amt = 50
  j done              # выход из блока case
case3:
  addi t0, zero, 3    # t0 = 3
  bne s0, t0, default # button = = 3?
  addi s1, zero, 100  # если да, amt = 100
  j done              # выход из блока case
default:
  add s1, zero, zero # amt=0
done:
```

Пример кода 6.18 ЦИКЛ `while`

Код на языке высокого уровня

```
// код выполняется до тех пор,
// пока x не примет такое значение,
// что 2^x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
pow = pow * 2;
x = x + 1;
}
```

Код на языке ассемблера RISC-V

```
# s0 = pow, s1 = x
       addi s0, zero, 1   # pow = 1
       add s1, zero, zero # x = 0

       addi t0, zero, 128 # t0 = 128
while: beq s0, t0, done   # pow = 128?
       slli s0, s0, 1     # pow = pow * 2
       addi s1, s1, 1     # x = x + 1
       j while            # повторение цикла
done
```

В ассемблерном коде в цикле `while` проверяется условие, противоположное условию, использованному на языке высокого уровня, аналогично тому, как это делается для оператора `if/else`. Если это противоположное условие истинно (в данном случае `s0 = 128`), цикл `while` завершается. В противном случае происходит умножение `pow` на 2 (используя сдвиг влево), увеличение `x` на 1 и переход обратно на начало цикла `while`.

Цикл `do/while` аналогичен циклу `while`, но *перед* проверкой условия он выполняет тело цикла *как минимум один раз*. **Пример кода 6.19** демонстрирует выполнение такого цикла. Обратите внимание, что, в отличие от предыдущих примеров, оператор условного перехода проверяет такое же условие, что и в коде высокого уровня.

Пример кода 6.19 ЦИКЛ `do/while`

Код на языке высокого уровня

```
// код выполняется до тех пор,
// пока x не примет такое значение,
// что 2^x = 128
int pow = 1;
int x = 0;

do {
  pow = pow * 2;
  x = x + 1;
} while (pow != 128);
```

Код на языке ассемблера RISC-V

```
# s0 = pow, s1 = x
       addi s0, zero, 1   # pow = 1
       add s1, zero, zero # x = 0

       addi t0, zero, 128 # t0 = 128
while: slli s0, s0, 1     # pow = pow * 2
       addi s1, s1, 1     # x = x + 1
       bne s0, t0, while  # pow = 128?
done:
```

Цикл for

Цикл `for`, как и цикл `while`, многократно выполняет участок кода до тех пор, пока условие цикла не станет ложным. При этом в цикле `for` используется счетчик цикла, который обычно хранит количество выполненных итераций цикла. Фактически цикл `for` – это удобное сокращение, объединяющее инициализацию счетчика, проверку условия прекращения цикла и изменение счетчика в одном месте. Обычно цикл `for` выглядит следующим образом:

```
for (инициализация; условие; операция цикла)
  оператор
```

Код инициализации выполняется до того, как цикл `for` начнется. Условие прекращения цикла проверяется в начале каждой итерации. Если условие не выполнено, цикл завершается. Операция цикла выполняется в конце каждой итерации.

Пример кода 6.20 складывает целые числа от 0 до 9. Счетчик цикла, в данном случае `i`, инициализируется нулем и увеличивается на единицу

в конце каждой итерации. Условие `i != 10` проверяется в начале каждой итерации. Итерация цикла `for` выполняется только тогда, когда условие истинно, т. е. когда значение `i` не равно 10, иначе цикл завершается. В нашем случае цикл `for` выполняется 10 раз. Циклы `for` могут быть реализованы и при помощи циклов `while`, но цикл `for` часто использовать удобнее. Обратите внимание, что этот пример также иллюстрирует использование операторов сравнения. Цикл на языке высокого уровня проверяет условие `(<)` для продолжения цикла, поэтому ассемблерный код проверяет противоположное условие `(>=)` для выхода из цикла.

Цикл `for` особенно полезен для доступа к большому количеству похожих данных, хранящихся в массивах памяти, о которых пойдет речь в следующем разделе.

Пример кода 6.20 ЦИКЛ `for`

Код на языке высокого уровня

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
}
```

Код на языке ассемблера RISC-V

```
# s0 = i, s1 = sum
       addi s1, zero, 0  # sum = 0
       addi s0, zero, 0  # i = 0
       addi t0, zero, 10 # t0 = 10
for:   bge s0, t0, done  # i >= 10?
       add s1, s1, s0    # sum = sum + i
       addi s0, s0, 1    # i = i + 1
       j for             # повторение цикла
done:
```

6.3.6. Массив

Массив удобно использовать для доступа к большому количеству однородных данных. Массив располагается в ячейках памяти со строго последовательными адресами и занимает непрерывный участок памяти. Каждый массив состоит из последовательности элементов одинакового размера, и каждый элемент массива имеет порядковый номер, называемый *индексом*. Количество элементов в массиве называется *длиной массива*. На **рис. 6.4** показан массив из 200 оценок в виде целых чисел, сохраненных в памяти. Адреса элементов массива в памяти последовательно увеличиваются на количество байтов в целом числе, т. е. на 4. Адрес нулевого элемента массива называется *базовым адресом* массива.

Рис. 6.4 Массив `scores[200]`, размещенный в памяти начиная с базового адреса 0x174300A0

В **примере кода 6.21** приведен алгоритм, который извлекает значение оценки из памяти, добавляет к ней 10 баллов и сохраняет обратно. Код для инициализации массива оценок в примере не показан. Предположим, что `s0` изначально равен базовому адресу мас-

сива 0x174300A0. Индекс массива – это переменная `i`, которая последовательно увеличивается на 1 при переходе к следующему элементу массива, поэтому мы умножаем ее на 4 и прибавляем к базовому адресу, чтобы получить правильный адрес элемента в памяти.

Пример кода 6.21 ДОСТУП К МАССИВУ С ПОМОЩЬЮ ЦИКЛА `for`

Код на языке высокого уровня

```
int i;
int scores[200];

for (i = 0; i < 200; i = i + 1)

scores[i] = scores[i] + 10;
```

Код на языке ассемблера RISC-V

```
# s0 = scores base address, s1 = i

  addi s1, zero, 0   # i = 0
  addi t2, zero, 200 # t2 = 200

for:
  bge s1, t2, done   # если i >= 200 завершить цикл
  slli t0, s1, 2     # t0 = i * 4
  add t0, t0, s0     # адрес scores[i]
  lw t1, 0(t0)       # t1 = scores[i]
  addi t1, t1, 10    # t1 = scores[i] + 10
  sw t1, 0(t0)       # scores[i] = t1
  addi s1, s1, 1     # i = i + 1
  j for              # повтор цикла
done:
```

Другие языки программирования, такие как Java, используют иные способы кодирования символов, в частности формат Unicode. В первых версиях стандарта Unicode для кодов символов отводилось 16 бит, что позволяло поддерживать диакритические знаки (ударения, умляуты и прочие) и разнообразные языки, в том числе азиатские. В современной версии Unicode определено более ста тысяч различных символов, и 16 бит уже недостаточно для кода произвольного символа. Это вынуждает отводить на каждый символ Unicode 32 бита памяти или использовать одно из представлений с переменной длиной, например UTF-16. Чтобы узнать больше о формате Unicode, посетите сайт www.unicode.org.

Байты и символы

Так как на англоязычной клавиатуре менее 128 символов, то символы английского языка обычно хранятся не в целых машинных словах, а в восьмибитовых байтах, каждый из которых способен хранить до 256 различных значений. Язык C использует тип данных `char` для представления байтов или символов.

В ранних компьютерах отсутствовало однозначное соответствие между байтами и символами английского языка, поэтому текстовый обмен между компьютерами был затруднителен. В 1963 году американская ассоциация по стандартизации опубликовала Американский стандартный код для обмена информацией (American Standard Code for Information Interchange, ASCII), в котором каждому символу было назначено уникальное значение байта[1].

[1] Тип `char` в языке C определен как целочисленный тип данных размером не менее 8 бит. На практике встречаются системы, где размер байта и, соответственно, типа `char` больше, чем 8 бит. Во избежание путаницы с размером байта иногда используют термин *октет*, означающий ровно 8 бит. Тип `char` в языке C может представлять либо знаковые, либо беззнаковые целые числа. Компилятор C вправе реализовать `char` и так, и иначе. Чтобы избавиться от неоднозначности, используйте вместо типа `char` либо тип `signed char`, либо тип `unsigned char`. – *Прим. перев.*

В **табл. 6.2** приведены коды для всех печатных символов. Значения ASCII приведены в шестнадцатеричной форме. Буквы верхнего и нижнего регистров отличаются на 0x20 (32).

Таблица 6.2 Кодировка ASCII

#	Символ	#	Символ	#	Символ	#	Символ	#	Символ	#	Символ
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Инструкции *загрузки байта* (`lb`), *загрузки байта без знака* (`lbu`) и *сохранения байта* (`sb`) обращаются к отдельным байтам в памяти. Инструкция `lb` дополняет байт знаковым битом, а инструкция `lbu` дополняет байт нулями, чтобы заполнить весь 32-битный регистр. Инструкция `sb` сохраняет самый младший байт 32-битного регистра по указанному байтовому адресу в памяти. Все три инструкции приведены на **рис. 6.5** с базовым адресом `s4`, равным 0xD0. Инструкция `lbu s1, 2(s4)` загружает байт из памяти по адресу 0xD2 в младший значащий байт `s1` и заполняет оставшиеся биты регистра нулями. Инструкция `lb s2, 3(s4)` загружает байт из памяти по адресу 0xD3 в младший значащий байт `s2` и заполняет оставшиеся разряды регистра знаковым битом, т. е. единицей. Инструкция `sb s3, 1(s4)` сохраняет младший байт `s3` (0x9B) в память по адресу 0xD1; она

Набор инструкций RISC-V также содержит инструкции загрузки и хранения *полуслов* `lh`, `lhu` и `sh`, которые оперируют 16-битными данными. Адреса памяти для этих инструкций должны быть выровнены по полуслову.

заменяет 0x42 на 0x9B. Никакие другие байты памяти не изменяются, а самые старшие значащие байты `s3` игнорируются.

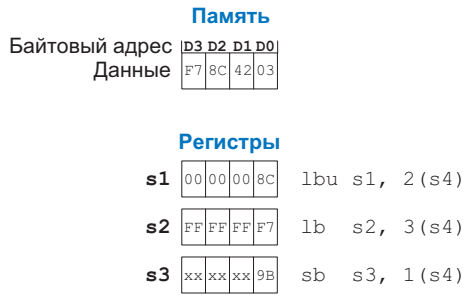


Рис. 6.5 Инструкции загрузки и сохранения байтов

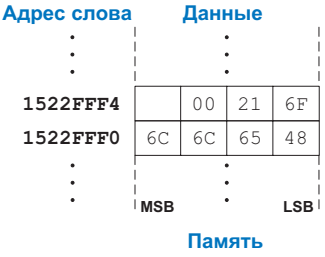


Рис. 6.6 Строка «Hello!», расположенная в памяти

Последовательность символов называют *строкой* (string). У строк переменная длина, поэтому языки программирования должны предоставлять какой-нибудь способ определения либо длины, либо конца строки. В языке C в конце строки указывается нулевой символ (0x00). Например, на **рис. 6.6** показана строка «Hello!» (0x48 65 6C 6C 6F 21 00), хранимая в памяти. Строка имеет длину 7 байт и занимает адреса от 0x1522FFF0 до 0x1522FFF6. Первый символ строки (H = 0x48) хранится по наименьшему адресу (0x1522FFF0).

Пример 6.2 ИСПОЛЬЗОВАНИЕ `lb` И `sb` ДЛЯ ДОСТУПА К МАССИВУ СИМВОЛОВ

Приведенный ниже код на языке программирования высокого уровня преобразует буквы, находящиеся в массиве символов из 10 элементов, из строчных в прописные путем вычитания 32 из каждого элемента массива. Преобразуйте этот код на язык ассемблера RISC-V. Не забудьте, что элементы массива теперь имеют размер 1 байт, а не 4 байта, поэтому соседние элементы имеют последовательные адреса. Будем считать, что `s0` уже содержит базовый адрес `chararray`.

```
// код высокого уровня
// chararray[10] был объявлен и инициализирован раньше
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

Решение

```
# код на языке ассемблера RISC-V
# s0 = базовый адрес chararray (инициализирован раньше), s1 = i
        addi s1, zero, 0    # i = 0
        addi t3, zero, 10   # t3 = 10
for:    bge s1, t3, done    # i >= 10 ?
        add t4, s0, s1      # t4 = адрес chararray[i]
        lb t5, 0(t4)        # t5 = chararray[i]
        addi t5, t5, -32    # t5 = chararray[i] - 32
```

```
        sb t5, 0(t4)        # chararray[i] = t5
        addi s1, s1, 1      # i = i + 1
        j for               # повторение цикла
done:
```

6.3.7. Вызовы функций

В языках высокого уровня обычно используют *функции*, или процедуры, для повторного использования часто выполняемого кода и для того, чтобы сделать программу модульной и читаемой. У функций есть входные параметры, называемые *аргументами*, и выходной результат, называемый *возвращаемым значением*. Функции должны вычислять возвращаемое значение, не вызывая неожиданных побочных эффектов.

Когда одна функция вызывает другую, вызывающая функция и вызываемая функция должны прийти к соглашению о том, где размещать аргументы и возвращаемое значение. Следуя соглашениям, принятым в архитектуре RISC-V, вызывающая функция обычно помещает до восьми аргументов в регистры от `a0` до `a7` перед вызовом функции, перед тем как произвести вызов, а вызываемая функция помещает возвращаемое значение в регистр `a0`, перед тем как завершить работу. Следуя этому соглашению, обе функции знают, где искать аргументы и куда возвращать значение, даже если вызывающая и вызываемая функции были разработаны разными людьми.

Вызываемая функция не должна вмешиваться в работу вызывающей функции. Это означает, что вызываемая функция должна знать, куда передать управление после завершения работы, и она не должна изменять значения любых регистров или памяти, которые нужны вызывающей функции. Вызывающая функция сохраняет *адрес возврата* в регистре адреса возврата `ra` (return address) в тот момент, когда она передает управление вызываемой функции путем выполнения инструкции *безусловного перехода с возвратом* `jal`. Вызываемая функция не должна изменять архитектурное состояние и содержимое памяти, от которых зависит вызывающая функция. В частности, вызываемая функция должна оставить неизменным содержимое сохраняемых регистров `s0-s11`, адрес возврата `ra` и *стек* – участок памяти, используемый для хранения временных переменных[1].

Коды ASCII развились из более ранних форм символьных кодировок. В 1838 году телеграфы начали использовать азбуку Морзе, то есть последовательность точек и тире, для передачи символов. В современной азбуке Морзе буквы A, B, C и D представляются как «.—», «—...», «—.—.» и «—..» соответственно. Количество и порядок точек и тире отличаются для каждой буквы, и часто встречающиеся буквы имеют более короткие коды, что повышает компактность кодировки. В 1874 году Жан Морис Эмиль Бодо изобрел 5-битный код, названный азбукой Бодо. В усовершенствованной азбуке Бодо—Мюррея буквы A, B, C и D были представлены как 00011, 11001, 01110 и 01001. Но 32 возможных вариантов этого 5-битного кода было недостаточно для всех английских символов, а 7-битной кодировки было достаточно. Таким образом, с развитием электронных средств связи 7-битная кодировка ASCII стала стандартом. На практике под символы ASCII обычно отводятся целые байты, а кодировку ASCII зачастую расширяют до восьми бит, что позволяет закодировать в одном байте 128 дополнительных символов, например символов другого языка.

Фактически RISC-V предоставляет два регистра для возвращаемого значения: `a0` и `a1`. Это позволяет возвращать 64-битные значения, такие как int64_t.

[1] Иными словами, если в вызываемой функции нужно изменить эти регистры, то необходимо сохранить их значения в каком-нибудь другом месте и восстановить перед возвратом из функции. – *Прим. перев.*

В этом разделе мы покажем, как вызывать функции и возвращаться из них, продемонстрируем, как функции получают доступ к входным аргументам и возвращают значение, а также то, как они используют стек для хранения временных переменных.

Вызовы и возвраты из функций

Архитектура RISC-V использует инструкцию *безусловного перехода с возвратом* `jal` для вызова функции и инструкцию безусловного перехода по регистру `jr` для возврата из функции. **Пример кода 6.22** демонстрирует главную функцию `main`, которая вызывает функцию `simple`. Здесь функция `main` является вызывающей, а `simple` – вызываемой. Функция `simple` не получает входных аргументов и ничего не возвращает, она просто передает управление обратно вызывающей функции. В **примере кода 6.22** слева от каждой инструкции RISC-V приведены их адреса в шестнадцатеричном формате.

Инструкции безусловного перехода с возвратом (`jal`) и безусловного перехода по регистру (`jr ra`) – две необходимые для вызова функций инструкции. В **примере кода 6.22** функция `main` вызывает функцию `simple` при помощи инструкции `jal`, которая выполняет две задачи: переходит на метку `simple` (0x0000051C) и сохраняет адрес возврата, которым является адрес инструкции, расположенной в памяти сразу после `jal` (в данном случае 0x00000304) в регистре адреса возврата `ra`. Программист может указать, в какой регистр будет записан адрес возврата, но по умолчанию это `ra`. Заметим, что код `jal simple` эквивалентен коду `jal ra, simple` и является предпочтительным с точки зрения стиля программирования. Функция `simple` немедленно завершается, выполняя инструкцию `jr ra`, которая переходит к адресу инструкции, содержащемуся в `ra`. Затем основная функция продолжает выполнение по этому адресу (0x00000304).

Адрес инструкции, выполняемой в данный момент, хранится в счетчике программ PC. Таким образом, следующий адрес инструкции обозначается как PC + 4.

Пример кода 6.22 ВЫЗОВ ФУНКЦИИ `simple`

Код на языке высокого уровня

```
int main() {
  simple();
  ...
}

// void означает, что функция
// не возвращает значение
void simple() {
  return;
}
```

Код на языке ассемблера RISC-V

```
0x00000300 main: jal simple  # вызов функции
0x00000304 ...
...        ...

0x0000051c simple: jr ra     # возврат
```

Входные аргументы и возвращаемые значения

В **примере кода 6.22** функция `simple` не очень-то полезна, потому что она не получает входных значений от вызывающей функции (`main`) и ничего не возвращает. По соглашениям, принятым в архитектуре RISC-V, функции используют регистры `a0-a7` для входных аргументов и `a0` для возвращаемого значения. В **примере кода 6.23** функция `diffofsums` вызывается с четырьмя аргументами и возвращает один результат. Мы решили сохранить локальную переменную `result` в `s3`. (О сохранении и восстановлении регистров мы расскажем немного позже.)

Следуя соглашениям RISC-V, вызывающая функция `main` помещает аргументы функции слева направо в регистры входных значений `a0-a7` перед вызовом функции. Вызываемая функция `diffofsums` сохраняет возвращаемое значение в регистре возврата `a0`. Если функции нужно передать более восьми аргументов, то дополнительные аргументы размещаются в стеке, использование которого мы обсудим далее.

Псевдоинструкции `j` и `jr` не являются частью набора инструкций, но удобны для программирования. Ассемблер RISC-V заменяет их настоящими инструкциями RISC-V. Ассемблер заменяет инструкцию `j target` на `jal zero, target`, которая выполняет переход и отбрасывает адрес возврата, записывая его в нулевой регистр; также ассемблер заменяет `jr ra` на `jalr zero, ra, 0`. Регистровая инструкция безусловного перехода с возвратом (`jalr`) похожа на `jal`, но она берет адрес назначения из регистра, опционально добавляемого к 12-битному непосредственному операнду со знаком. Например, `jalr ra, s1, 0x4C` выполняет переход на адрес `s1 + 0x4C` и помещает увеличенное на 4 значение программного счетчика в `ra`.

Пример кода 6.23 ВЫЗОВ ФУНКЦИИ С АРГУМЕНТАМИ И ВОЗВРАЩАЕМЫМ ЗНАЧЕНИЕМ

Код на языке высокого уровня

```
int main(){
  int y;
  . . .



  y = diffofsums(2, 3, 4, 5);
  . . .
}

int diffofsums(int f, int g, int h,
               int i){
  int result;

  result = (f + g) - (h + i);

  return result;
}
```

Код на языке ассемблера RISC-V

```
# s7 = y
main:
   . . .
   addi a0, zero, 2  # argument 0 = 2
   addi a1, zero, 3  # argument 1 = 3
   addi a2, zero, 4  # argument 2 = 4
   addi a3, zero, 5  # argument 3 = 5
   jal diffofsums    # вызов функции
   add s7, a0, zero  # y = возвращаемое значение
   . . .
# s3 = result
diffofsums:
   add t0, a0, a1    # t0 = f+g
   add t1, a2, a3    # t1 = h+i
   sub s3, t0, t1    # result = (f+g)-(h+i)
   add a0, s3, zero  # возвращаемое значение в a0
   jr ra             # возврат в место вызова
```

Стек обычно хранится в памяти в перевернутом виде так, что вершине стека фактически соответствует наименьший адрес памяти, и стек растет вниз по направлению к еще более меньшим адресам памяти. Такой стек называется *нисходящим* (Descending stack). Некоторые архитектуры также допускают *восходящий* стек (Ascending stack), который растет вверх, в направлении более высоких адресов памяти. *Указатель стека* (sp) обычно указывает на самый верхний элемент в стеке (т. е. последний элемент, который был помещен в стек); такой стек называется *полным* (Full stack). Некоторые архитектуры, такие как, например, ARM, также позволяют использовать *пустой стек* (Empty stack), в котором sp указывает на слово, следующее за вершиной стека (т. е. место, куда будет сохранен следующий элемент, помещенный в стек). Архитектура RISC-V определяет *полный нисходящий стек*, который мы будем использовать в этой главе. Это стандартный способ, с помощью которого функции передают переменные и используют стек, чтобы библиотеки, разработанные разными компиляторами, могли взаимодействовать друг с другом.

Стек

Стек (stack) – это участок памяти для хранения локальных переменных функции. Стек расширяется (занимает больше памяти), если процессору нужно больше места, и сужается (занимает меньше памяти), если процессору больше не нужны сохраненные там переменные. Прежде чем объяснять, как функции используют стек для хранения временных значений, мы объясним, как стек работает.

Стек является очередью, работающей в режиме «*последний пришел – первый ушел*» (LIFO, last-in-first-out). Как и в стопке тарелок, последний элемент, помещенный (push) на стек (верхняя тарелка), будет первым элементом, который с него снимут (извлекут, pop). Каждая функция может выделить память в стеке для хранения локальных переменных, и она же должна освободить выделенную память перед возвратом. *Вершина стека* (top of the stack) – это память, которая была выделена последней. Так же как стопка тарелок растет вверх в пространстве, размер стека в архитектуре RISC-V увеличивается в памяти. Стек расширяется в сторону младших адресов по мере выделения нового места в памяти для программы (функции).

На **рис. 6.7** изображен стек. *Регистр указателя стека* (sp, от англ. *stack pointer*) – это специальный регистр, который указывает на вершину стека. *Указатель* (pointer) – специальное имя для обычного адреса памяти. Он указывает на данные программы, то есть хранит их адрес. Например, на **рис. 6.7 (а)** указатель стека sp содержит адрес 0xBEFFFAE8 и указывает на значение данных 0xAB000001.

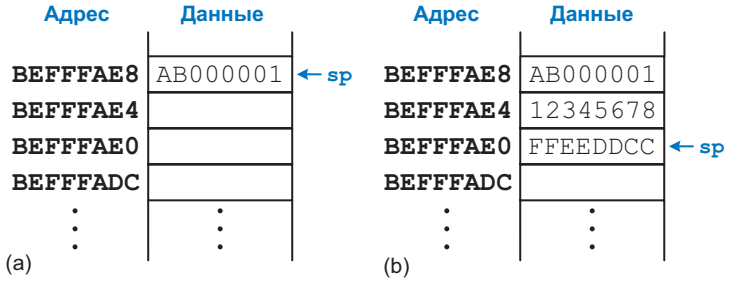


Рис. 6.7 Стек (а) до расширения и (b) после расширения

Указатель стека (sp) изначально равен большему адресу памяти, после чего значение адреса по необходимости уменьшается для увеличения доступного программе места. На **рис. 6.7 (b)** изображен стек, расширяющийся для того, чтобы выделить два дополнительных слова данных для хранения временных переменных. Для этого значение регистра sp уменьшается на 8 и становится равным 0xBEFFFAE0. Два дополнительных слова данных, 0x12345678 и 0xFFEEDDCC, временно размещаются в стеке.

Одно из важных применений стека – сохранение и восстановление значений регистров, используемых внутри функции. Вспомним, что функция должна производить вычисления и возвращать значения, но не должна приводить к неожиданным побочным эффектам. В частности, она не должна менять значения никаких регистров, кроме регистра a0, содержащего возвращаемое значение. Функция diffofsums в **примере кода 6.23** нарушает это правило, поскольку изменяет регистры t0, t1 и s3. Если бы функция main использовала эти регистры до вызова diffofsums, то содержимое этих регистров было бы повреждено вызовом данной функции.

Чтобы решить эту проблему, функция сохраняет значения регистров в стеке, перед тем как изменить их, а затем восстанавливает их из стека перед возвратом. В частности, она выполняет следующие шаги:

1) выделяет пространство в стеке для сохранения значений одного или нескольких регистров;
2) сохраняет значения регистров в стеке;
3) выполняет функцию, используя регистры;
4) восстанавливает исходные значения регистров из стека;
5) освобождает пространство в стеке.

В **примере кода 6.24** приведена улучшенная версия функции diffofsums, которая сохраняет и восстанавливает регистры t0, t1 и s3. На **рис. 6.8** показан стек до, во время и после вызова функции diffofsums из **примера кода 6.24**. Стек начинается с адреса 0xBEF0F0FC. Функция diffofsums выделяет пространство в стеке для трех слов, уменьшая указатель стека sp на 12. Затем она сохраняет текущие значения t0, t1 и s3 во вновь выделенном пространстве. Дальше выполняется остальная часть функции, которая меняет значения этих трех регистров. В конце своего выполнения функция diffofsums восстанавливает значения регистров t0, t1 и s3 из стека, освобождает пространство стека и возвращается в main. Когда функция выполняет возврат, в регистре a0 содержится результат, но другие побочные эффекты отсутствуют: t0, t1, s3 и sp имеют те же значения, что и до вызова функции.

Пример кода 6.24 ФУНКЦИЯ, СОХРАНЯЮЩАЯ РЕГИСТРЫ В СТЕКЕ

Код на языке высокого уровня

```
int diffofsums(int f, int g,
               int h, int i){
  int result;

  result = (f + g) - (h + i);

  return result;
}
```

Код на языке ассемблера RISC-V

```
# s3 = result
diffofsums:
  addi sp, sp, -12 # выделить в стеке место
                   # для хранения трех регистров
  sw s3, 8(sp)     # сохранить s3 в стеке
  sw t0, 4(sp)     # сохранить t0 в стеке
  sw t1, 0(sp)     # сохранить t1 в стеке
  add t0, a0, a1   # t0 = f + g
  add t1, a2, a3   # t1 = h + i
  sub s3, t0, t1   # result = (f + g) - (h + i)
  add a0, s3, zero # поместить возвращаемое значение в a0
  lw s3, 8(sp)     # восстановить s3 из стека
  lw t0, 4(sp)     # восстановить t0 из стека
  lw t1, 0(sp)     # восстановить t1 из стека
  addi sp, sp, 12  # освободить пространство стека
  jr ra            # возврат в место вызова
```

Рис. 6.8 Стек: (a) до, (b) во время и (c) после вызова функции `diffofsums`

Сохранение значения регистра в стеке называется помещением (push) регистра в стек. Восстановление значения регистра из стека называется извлечением (pop) регистра из стека.

Пространство стека, которое функция выделяет для себя, называется *фреймом стека*. Фрейм стека функции `diffofsums` состоит из трех слов. Из принципа модульности следует, что каждая функция должна иметь доступ только к своему собственному фрейму стека, не имея возможности повредить фреймы, принадлежащие другим функциям.

Оберегаемые регистры

В **примере кода 6.24** предполагается, что все временные регистры (`t0`, `t1` и `s3`) должны быть сохранены и восстановлены. Если вызывающая функция не использует эти регистры, то усилия по сохранению и восстановлению их значений тратятся впустую. Чтобы избежать этих

издержек, в архитектуре RISC-V регистры разделены на две категории: *оберегаемые* (preserved) и *необерегаемые* (nonpreserved). Оберегаемые регистры должны содержать одни и те же значения до начала работы и после окончания работы вызываемой функции, поскольку вызывающая функция ожидает, что сохраненные значения регистров останутся прежними после завершения вызова.

Оберегаемые регистры включают `s0-s11` (отсюда их название: saved, сохраняемые), `sp` и `ra`. Необерегаемые регистры, также называемые *временными регистрами*, – это регистры от `t0` до `t6` (отсюда их название: temporary, временные) и регистры аргументов `a0-a7`. Функция должна сохранять и восстанавливать любые оберегаемые регистры, с которыми она собирается работать, но может свободно менять значения необерегаемых регистров.

В **примере кода 6.25** показана улучшенная версия функции `diffofsums`, которая сохраняет в стеке только регистр `s3`. Регистры `t0` и `t1` являются необерегаемыми регистрами, поэтому их сохранять не обязательно.

Пример кода 6.25 ФУНКЦИЯ, СОХРАНЯЮЩАЯ ОБЕРЕГАЕМЫЕ РЕГИСТРЫ В СТЕКЕ

Код на языке ассемблера RISC-V

```
# s3 = result
diffofsums:
  addi sp, sp, -4   # выделить в стеке место для хранения одного регистра
  sw s3, 0(sp)      # сохранить s3 в стеке
  add t0, a0, a1    # t0 = f + g
  add t1, a2, a3    # t1 = h + i
  sub s3, t0, t1    # result = (f + g) - (h + i)
  add a0, s3, zero  # поместить возвращаемое значение в a0
  lw s3, 0(sp)      # восстановить s3 из стека
  addi sp, sp, 4    # освободить пространство стека
  jr ra             # возврат в место вызова
```

Вспомним, что когда одна функция вызывает другую, то первая называется *вызывающей функцией*, а вторая – *вызываемой*. Вызываемая функция должна сохранять и восстанавливать любые оберегаемые регистры, которые собирается использовать, но может свободно изменять любые необерегаемые регистры. Следовательно, если вызывающая функция держит актуальные данные в необерегаемых регистрах, она должна сохранять необерегаемые регистры, перед тем как вызывать другую функцию, а затем их восстанавливать. По этой причине оберегаемые регистры также называют *сохраняемыми вызываемой функцией*, а необерегаемые регистры называют *сохраняемыми вызывающей функцией*.

В **табл. 6.3** приведены все оберегаемые регистры. Соглашение о том, какие регистры следует оберегать, является частью стандартного согла-

шения о вызовах[1] для архитектуры RISC-V и не относится к самой архитектуре.

Таблица 6.3 Оберегаемые и необерегаемые регистры и память

Оберегаемые (сохраняет вызываемая функция)	Необерегаемые (сохраняет вызывающая функция)
Сохраняемые регистры: `s0-s11`	Временные регистры: `t0-t6`
Адрес возврата: `ra`	Регистры аргументов: `a0-a7`
Указатель стека: `sp`	
Содержимое стека до указателя	Содержимое стека после указателя

Регистры `s0-s11` обычно используются для хранения локальных переменных внутри функции, поэтому их необходимо сохранить. Регистр `ra` также следует сохранять, чтобы вызываемая функция знала, куда возвращаться. Регистры `t0-t6` используются для хранения временных результатов. Вычисления, использующие временные результаты, обычно завершаются до того, как вызывается функция, поэтому эти регистры не оберегаются, а необходимость сохранять их в вызывающей функции возникает крайне редко. Регистры `a0-a7` часто перезаписываются в процессе вызова функции, поэтому вызывающая функция должна сохранять их, если эти значения могут понадобиться ей после завершения вызванной функции.

Стек выше указателя стека автоматически остается в сохранности, если только вызываемая функция не осуществляет запись в память по адресам выше `sp`. При таком подходе она не меняет *фреймы стека* (stack frames) других функций. Сам указатель стека остается в сохранности, потому что вызываемая функция перед завершением работы освобождает свой фрейм стека, прибавляя к `sp` то же значение, которое вычла из него в начале.

Проницательный читатель или оптимизирующий компилятор может заметить, что локальная переменная `result` функции `diffofsums` немедленно возвращается и не используется ни для каких целей. Следовательно, мы можем исключить переменную и просто сохранить результат вычисления непосредственно в регистре `a0`, устраняя необходимость выделять пространство в фрейме стека и перемещать результат из `s3` в `a0`. В **примере кода 6.26** приведена еще более оптимизированная версия функции `diffofsums`.

[1] https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

Пример кода 6.26 ОПТИМИЗИРОВАННАЯ ФУНКЦИЯ `diffofsums`

Код на языке ассемблера RISC-V

```
# s3 = result
diffofsums:
  addi sp, sp, -4   # выделить в стеке место для хранения одного регистра
  sw s3, 0(sp)      # сохранить s3 в стеке
  add t0, a0, a1    # t0 = f + g
  add t1, a2, a3    # t1 = h + I
  sub s3, t0, t1    # result = (f + g) - (h + i)
  add a0, s3, zero  # поместить возвращаемое значение в a0
  lw s3, 0(sp)      # восстановить s3 из стека
  addi sp, sp, 4    # освободить пространство стека
  jr ra             # возврат в место вызова
```

Вызовы нелистовых функций

Функция, которая не вызывает другие функции, называется *листовой* (leaf function); пример – функция `diffofsums`. Функция, которая вызывает другие функции, называется *нелистовой функцией* (nonleaf function). Как было замечено ранее, нелистовые функции устроены более сложно, потому что перед вызовом других функций им приходится сохранять необеpегаемые регистры в стеке и затем восстанавливать эти регистры. В частности, они должны соблюдать следующие правила.

Правило сохранения вызывающей функции: перед вызовом другой функции вызывающая функция должна сохранить все необеpегаемые регистры (`t0-t6` и `a0-a7`), которые ей понадобятся после завершения вызова. После вызова она должна восстановить эти регистры до того, как они понадобятся.

Правило сохранения вызываемой функции: прежде чем вызываемая функция изменит какой-либо из обеpегаемых регистров (`s0-s11` и `ra`), она должна сохранить их. Непосредственно перед возвратом из вызова она должна восстановить эти регистры.

В **примере кода 6.27** показаны нелистовая функция `f1` и листовая функция `f2`, а также все необходимые операции сохранения регистров. Функция `f1` сохраняет `i` в регистре `s4` и `x` в регистре `s5`; функция `f2` сохраняет `r` в регистре `s4`. Функция `f1` использует обеpегаемые регистры `s4`, `s5` и `ra`, поэтому сначала помещает их в стек в соответствии с правилом сохранения вызываемой функции. Она использует `t3` для хранения промежуточного результата `(a - b)`, поэтому ей не нужно сохранять регистр с результатом этого вычисления. Прежде чем вызвать `f2`, функция `f1` сохраняет `a0` и `a1` в стек в соответствии с правилом сохранения вызывающей функции, потому что это необеpегаемые регистры, которые `f2` может изменить, и они все равно понадобятся функции `f1` после вызова. Содержимое регистра `ra` изменяется, поскольку оно перезаписывается результатом вызова `f2`. Хотя `t3` также является необеpегаемым регист-

ром, который функция f2 может перезаписать, функция f1 больше не нуждается в этом регистре, поэтому его не нужно сохранять. Затем f1 передает аргумент f2 в регистре a0, выполняет вызов функции и получает результат в a0. Затем f1 восстанавливает значения регистров a0 и a1, потому что они все еще нужны для дальнейших вычислений. Когда функция f1 завершает свою работу, она помещает возвращаемое значение в a0, восстанавливает регистры s4, s5, ra и sp и возвращается в точку вызова. Функция f2 сохраняет и восстанавливает регистр s4 и указатель стека sp в соответствии с правилом сохранения вызываемой функции.

Пример кода 6.27 ВЫЗОВ НЕЛИСТОВОЙ ФУНКЦИИ

Код на языке высокого уровня	Код на языке ассемблера RISC-V
`int f1(int a, int b) {`	`# a0 = a, a1 = b, s4 = i, s5 = x`
`  int i, x;`	`f1:`
	`  addi  sp, sp, -12  # место в стеке для 3 регистров`
	`  sw    ra, 8(sp)    # сохранение оберегаемых регистров f1`
	`  sw    s4, -4(sp)`
	`  sw    s5, 0(sp)`
`  x = (a + b)*(a - b);`	`  add   s5, a0, a1   # x = (a + b)`
	`  sub   t3, a0, a1   # temp = (a - b)`
	`  mul   s5, s5, t3   # x = x * temp = (a + b) * (a - b)`
	`  addi  s4, zero, 0  # i = 0`
`  for (i = 0; i < a; i++)`	`for:`
	`  bge   s4, a0, return # если i >= a, то выход из цикла`
	`  addi  sp, sp, -8   # место в стеке для 2 регистров`
	`  sw    a0, 4(sp)    # сохранение необерегаемых регистров в стек`
`    x = x + f2(b + i);`	`  sw    a1, 0(sp)`
	`  add   a0, a1, s4   # аргумент равен b + i`
	`  jal   f2           # call f2(b + i)`
	`  add   s5, s5, a0   # x = x + f2(b + i)`
	`  lw    a0, 4(sp)    # восстановление необерегаемых регистров`
	`  lw    a1, 0(sp)`
	`  addi  sp, sp, 8`
	`  addi  s4, s4, 1    # i++`
`    return x;`	`  j     for          # продолжение цикла`
`  }`	`return:`
	`  add   a0, zero, s5 # возвращаемое значение x`
	`  lw    ra, 8(sp)    # восстановление оберегаемых регистров`
	`  lw    s4, 4(sp)`
	`  lw    s5, 0(sp)`
	`  addi  sp, sp, 12   # восстановление указателя стека sp`
	`  jr    ra           # возврат из f1`
`int f2(int p) {`	`# a0 = p, s4 = r`
`  int r;`	`f2:`
	`  addi sp, sp, -4    # сохранение оберегаемых регистров f2`
`  r = p + 5;`	`  sw s4, 0(sp)`
`  return r + p;`	`  addi s4, a0, 5     # r = p + 5`
`}`	`  add a0, s4, a0     # возвращаемое значение r + p`
	`  lw s4, 0(sp)       # восстановление оберегаемых регистров`
	`  addi sp, sp, 4     # восстановление указателя стека sp`
	`  jr ra              # возврат из f2`

При внимательном рассмотрении можно заметить, что функция `f2` не изменяет регистр `a1`, поэтому функция `f1` не обязана сохранять и восстанавливать его. Однако компилятор не всегда может уверенно определить, какие несохраненные регистры будут затронуты во время вызова функции. По этой причине простые компиляторы всегда заставляют вызывающую функцию сохранять и восстанавливать любые необеременные регистры, которые ей понадобятся после вызова. Оптимизирующий компилятор способен заметить, что `f2` является листовой функцией и может выделить для `r` необеременный регистр, избегая необходимости сохранять и восстанавливать `s4`. На **рис. 6.9** показано состояние стека во время выполнения функций. В этом примере указатель стека начинает с адреса 0xBEF7FF0C.

> Нелистовая функция перезаписывает регистр `ra`, когда она вызывает другую функцию с помощью команды `jal`. Следовательно, нелистовая функция всегда должна сохранять `ra` в своем стеке и восстанавливать его перед возвратом.

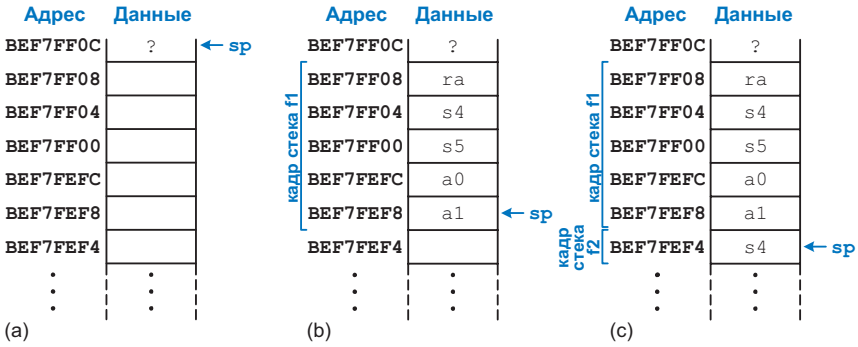


Рис. 6.9 Стек: (a) перед вызовом функции, (b) во время вызова `f1` и (c) во время вызова `f2`

Рекурсивные вызовы функций

Рекурсивная функция – это нелистовая функция, вызывающая сама себя. Рекурсивные функции ведут себя одновременно как вызывающая и вызываемая, поэтому должны сохранять как оберегаемые, так и необеременные регистры. Например, функция вычисления факториала может быть реализована как рекурсивная функция. Вспомним, что

$$factorial(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$$

В **примере кода 6.28** показана функция `factorial`, записанная в рекурсивном представлении как

$$factorial(n) = n \times factorial(n-1).$$

Факториал от 1 – это просто 1. Для удобства обращения к адресам программы мы предполагаем, что она начинается с адреса 0x8500. Факториал не является листовой функцией и, согласно правилу сохране-

ния вызываемой функции, должен сохранять `ra`. Кроме того, функции `factorial` потребуется значение `n` после вызова самой себя, поэтому, согласно правилу сохранения вызывающей функции, она должна сохранить `a0`. Таким образом, она начинает с размещения регистров `ra` и `a0` в стеке.

Затем она проверяет условие $n \leq 1$, и если это так, то помещает возвращаемое значение 1 в `a0`, восстанавливает указатель стека и возвращается к точке вызова. В этом случае нет необходимости восстанавливать регистр `ra`, потому что он не изменялся. Если $n > 1$, функция рекурсивно вызывает `factorial(n – 1)`. Затем она восстанавливает значение `n` и регистр адреса возврата (`ra`) из стека, выполняет умножение и возвращает результат. Обратите внимание на небольшую хитрость: функция восстанавливает `n` в `t1`, чтобы не перезаписать возвращаемое значение. Инструкция умножения (`mul a0, t1, a0`) перемножает `n` (`t1`) и возвращенное значение `a0` и помещает результат в `a0`.

Пример кода 6.28 РЕКУРСИВНЫЙ ВЫЗОВ ФУНКЦИИ `factorial`

Код на языке высокого уровня

```
int factorial(int n) {
  if (n <= 1)
    return 1;

  else
    return (n*factorial(n-1));
}
```

Код на языке ассемблера RISC-V

```
0x8500 factorial: addi sp, sp, -8   # выделить место
                                    # в стеке под a0, ra
0x8504            sw   a0, 4(sp)    # сохранить a0
0x8508            sw   ra, 0(sp)    # сохранить ra
0x850C            addi t0, zero, 1  # temporary = 1
0x8510            bgt  a0, t0, else # если n > 1, перейти
                                    # к else
0x8514            addi a0, zero, 1  # иначе возвратить 1
0x8518            addi sp, sp, 8    # восстановить sp
0x851C            jr   ra           # возврат
0x8520 else:      addi a0, a0, -1   # n = n – 1
0x8524            jal  factorial    # рекурсивный вызов
0x8528            lw   t1, 4(sp)    # восстановить n в t1
0x852C            lw   ra, 0(sp)    # восстановить ra
0x8530            addi sp, sp, 8    # восстановить sp
0x8534            mul  a0, t1, a0   # a0 = n *
                                    # factorial(n – 1)
0x8538            jr   ra           # возврат
```

Для наглядности мы сохраняем регистры в начале вызова функции. Оптимизирующий компилятор может заметить, что нет необходимости сохранять `a0` и `ra`, когда $n \leq 1$, и организовать сохранение регистров в стеке только в блоке `else` функции.

На **рис. 6.10** показан стек в процессе выполнения функции `factorial(3)`. Мы предполагаем, что `sp` изначально указывает на 0xFF0 (старшие биты адреса равны 0), как показано на **рис. 6.10 (а)**. Функция создает фрейм стека из двух слов для хранения `n` (`a0`) и `ra`. При первом вызове `factorial` сохраняет `a0` (содержащий `n` = 3) по адресу 0xFEC

и ra по адресу 0xFE8, как показано на **рис. 6.10 (b)**. Затем функция меняет a0 на n = 2 и рекурсивно вызывает factorial(2), при этом значение ra меняется на 0x8528. При втором вызове функция сохраняет a0 (содержащий n = 2) по адресу 0xFE4 и ra по адресу 0xFE0. В этот раз мы знаем, что ra содержит 0x8528. Затем функция меняет a0 на n = 1 и рекурсивно вызывает factorial(1). При третьем вызове она сохраняет a0 (содержащий n = 1) по адресу 0xFDC и ra по адресу 0xFD8. На этот раз ra снова содержит 0x8528. Третий вызов функции factorial возвращает значение 1 в a0 и освобождает фрейм стека перед возвратом ко второму вызову. Второй вызов восстанавливает значение n = 2 (в t1), восстанавливает значение 0x8528 в ra (так получилось, что он уже содержит это значение), освобождает фрейм стека и возвращает a0 = 2 × 1 = 2 первому вызову. Первый вызов восстанавливает n = 3, восстанавливает адрес возврата к вызывающей функции в ra, освобождает фрейм стека и возвращает a0 = 3 × 2 = 6. На **рис. 6.10** показан стек после завершения рекурсивно вызванных функций. Когда функция factorial возвращает управление вызвавшей ее функции, указатель стека находится в своем начальном положении (0xFF0), содержимое стека выше значения указателя стека не менялось и все оберегаемые регистры содержат свои начальные значения. Регистр a0 содержит результат вычисления 6.

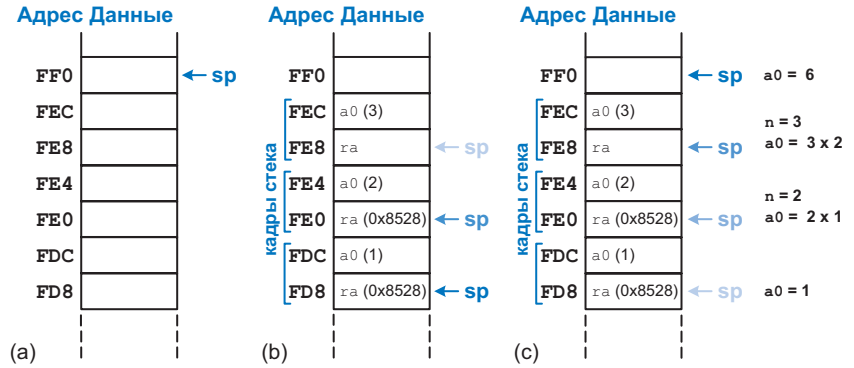


Рис. 6.10 Стек до (a), после последнего рекурсивного вызова (b) и после завершения (c) вызова функции factorial при *n* = 3

Дополнительные аргументы и локальные переменные

У функций может быть более восьми входных аргументов и слишком много локальных переменных для хранения в оберегаемых регистрах. Для хранения этих временных значений используют стек. Следуя соглашениям архитектуры RISC-V, если функция имеет более восьми аргументов, то первые восемь передаются в регистры аргументов (a0–a7) как обычно. Дополнительные аргументы передаются в стеке, прямо над ука-

зателем стека `sp`. *Вызывающая* функция должна расширить свой стек, чтобы выделить место для дополнительных аргументов. На **рис. 6.11 (a)** показан стек вызывающей функции при вызове функции, принимающей более восьми аргументов.

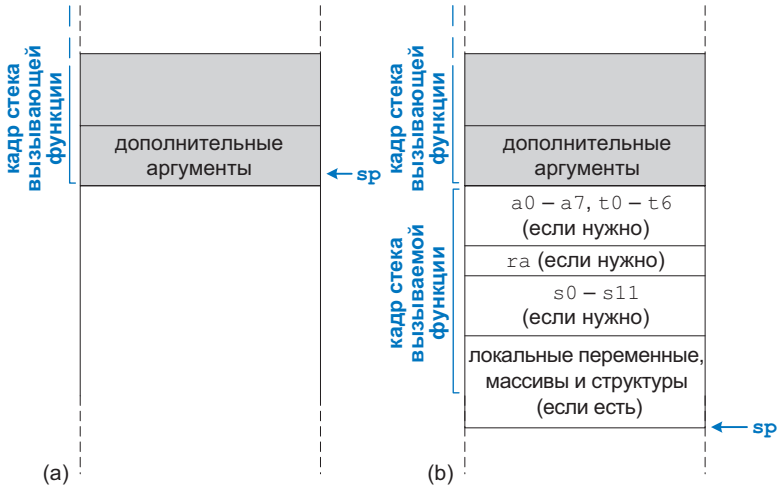


Рис. 6.11 Использование расширенного стека перед вызовом (a) и после вызова (b)

Функция также может объявлять локальные переменные или массивы. С *локальными* переменными, объявленными внутри функции, может работать только сама эта функция. Локальные переменные хранятся в регистрах `s0-s11`; если локальных переменных слишком много, их можно хранить во фрейме стека функции. В стеке также хранятся локальные массивы и структуры.

Некоторые функции также используют *указатель фрейма*, который указывает на *нижнюю* часть активного фрейма стека — т. е. фрейма стека выполняющейся функции. По соглашению этот адрес хранится в регистре `fp` (`x8`), который тоже является оберегаемым регистром.

На **рис. 6.11 (b)** показана структура стека вызываемой функции. Фрейм стека содержит аргументы самой функции, адрес возврата и любые оберегаемые регистры, которые функция может менять. Он также содержит локальные массивы и любые дополнительные локальные переменные. Если у вызываемой функции более четырех аргументов, она находит их во фрейме стека вызывающей функции. Доступ к дополнительным аргументам — это единственный случай, когда функции позволено читать данные из стека за пределами собственного фрейма.

6.3.8. Псевдокоманды

Прежде чем мы покажем, как преобразовать ассемблерный код в единицы и нули машинного кода, давайте вернемся к псевдокомандам. В архитектуре RISC-V размер команд и сложность аппаратного обеспечения

минимизированы путем использования лишь небольшого количества команд. Тем не менее RISC-V определяет псевдокоманды, которые на самом деле не являются частью набора команд, но часто используются программистами и компиляторами. При преобразовании в машинный код псевдокоманды транслируются в одну или несколько команд RISC-V. Например, мы уже обсуждали псевдокоманду безусловного перехода (`j`), которая преобразуется в инструкцию безусловного перехода с возвратом (`jal`) с регистром `x0` в качестве регистра-назначения, то есть адрес возврата не сохраняется. Мы также отметили, что логическая операция NOT (НЕ) может быть выполнена с помощью операции XOR (ИСКЛЮЧАЮЩЕЕ ИЛИ) между исходным операндом и регистром, содержащим единицы во всех разрядах.

Псевдокоманду `nop` обычно используют для формирования точных задержек при выполнении программы.

Таблица 6.4 содержит примеры псевдокоманд и соответствующих им команд RISC-V. Например, псевдокоманда перемещения `mv` копирует содержимое одного регистра в другой регистр. Псевдокоманда `li` загружает в регистр 32-битную константу, используя комбинацию инструкций `lui` и `addi`. Если константа помещается в 12 бит, песвдокоманда `li` транслируется в инструкцию `addi`. Псевдокоманда «нет операции» (`nop`) не выполняет никаких действий. После ее выполнения счетчик команд увеличивается на 4, но никакие другие регистры и содержимое памяти не изменяются. Псевдокоманда `call` выполняет вызов функции.

Таблица 6.4 Псевдокоманды и соответствующие инструкции RISC-V

Псевдокоманда	Инструкции RISC-V	Описание	Операция
`j label`	`jal zero, label`	Безусловный переход	`PC = label`
`jr ra`	`jalr zero, ra, 0`	Переход по адресу в регистре	`PC = ra`
`mv t5, s3`	`addi t5, s3, 0`	Перемещение значения из одного регистра в другой	`t5 = t3`
`not s7, t2`	`xori s7, t2, -1`	Дополнение до единицы	`s7 = ~t2`
`nop`	`addi zero, zero, 0`	Пустая операция	
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`	Загрузка 12-битной константы	`s8 = 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`	Загрузка 32-битной константы	`s8 = 0x56789DEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`	Переход, если >	`if (s1 > t3), PC = L3`
`bgez t2, L7`	`bge t2, zero, L7`	Переход, если ≥	`if (t2 ≥ 0), PC = L7`
`call L1`	`jal L1`	Вызов ближней функции	`PC = L1, ra = PC + 4`
`call L5`	`auipc ra, imm`$_{31:12}$ `jalr ra, ra, imm`$_{11:0}$	Вызов дальней функции	`PC = L5, ra = PC + 4`
`ret`	`jalr zero, ra, 0`	Возврат из функции	`PC = ra`

Если это вызов ближайшей функции, `call` транслируется в инструкцию `jalr`. Но если функция расположена далеко, вызов переводится в две инструкции RISC-V: `auipc` и `jalr`. Например, инструкция `auipc s1, 0xABCDE` добавляет к программному счетчику PC значение 0xABCDE000 и помещает результат в `s1`. Допустим, исходное значение PC = 0x02000000, тогда в регистр `s1` будет помещен результат 0xADCDE000. Затем инструкция `jalr ra, s1, 0x730` выполняет переход к адресу `s1` + `0x730` (0xADCDE730) и помещает в регистр `ra` значение PC + 4. Псевдокоманда `ret` выполняет возврат из функции и транслируется в `jalr x0, ra, 0`.

В **табл. B.7** в **приложении B** перечислены наиболее распространенные псевдокоманды набора инструкций RV32I.

6.4. Машинный язык

Язык ассемблера удобен для чтения человеком, но цифровые схемы понимают только нули и единицы. Поэтому программу, написанную на языке ассемблера, переводят из последовательности мнемоник в последовательность нулей и единиц, которую называют *машинным языком*. В этом разделе мы обсуждаем машинный язык RISC-V и трудоемкий процесс трансляции из ассемблера в машинный язык.

Для простоты нужно придерживаться единообразия, и наиболее единообразным представлением команд в машинном языке было бы такое, где каждая команда занимала бы ровно одно слово памяти. Длина команд в архитектуре RISC-V составляет 32 бита, при этом некоторые из них используют только часть из этих битов. И хотя можно было бы сделать длину команд переменной, это излишне усложнило бы архитектуру. Ради простоты также следовало бы применять единственный формат инструкции, но это слишком жесткое ограничение. Это позволяет нам представить последнее правило хорошей разработки.

Четвертое правило хорошей разработки:
она требует хороших компромиссов.

Любопытно, что тип *S/B* не называется *B/S*.

В архитектуре RISC-V в качестве компромисса используются четыре формата инструкций: типа *R*, типа *I*, типа *S/B* и типа *U/J*. Такое небольшое количество форматов обеспечивает единообразие инструкций и, как следствие, их более простую аппаратную реализацию. При этом разные форматы позволяют учитывать различные потребности инструкций, как, например, необходимость хранить большие константы внутри них. Инструкции типа *R* (регистровые), такие как `add s0, s1, s2`, используют три регистровых операнда. Инструкции типа *I* (immediate, непосредственные), такие как `addi s3, s4, 42`, и инструкции типа *S/B* (store/branch, сохранение

слова в памяти / условный переход), такие как `sw a0, 4(sp)` или `beq a0, a1, L1`, используют два регистра и 12- или 13-битную константу. Инструкции типа *U/J* (upper immediate/jump, старшие разряды константы / безусловный переход), такие как `jal ra, factorial`, работают с одним регистром и 20- или 21-битной константой. В этом разделе мы опишем все вышеупомянутые форматы машинных инструкций RISC-V и покажем, как они кодируются в двоичном формате. Приложение B содержит краткий справочник по всем инструкциям набора инструкций RV32I.

6.4.1. Инструкции типа *R*

Инструкции типа R используют три регистра в качестве операндов: два регистра-источника и один регистр-назначение. На **рис. 6.12** приведен машинный формат команд типа *R*. 32-битная команда состоит из шести полей: `funct7`, `rs2`, `rs1`, `funct3`, `rd` и `op`. Каждое поле занимает от трех до семи бит.

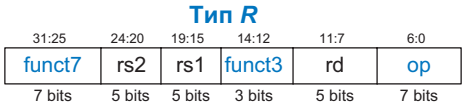


Рис. 6.12 Формат инструкции типа *R*

Операция, выполняемая командой, закодирована в трех полях, выделенных синим цветом: 7-битном поле `op` (также называемом `opcode`, или кодом операции) и 7- и 3-битными полями `funct7` и `funct3` (также называемыми функциями). Конкретная операция типа *R* определяется полями `op` и `funct`. Эти биты вместе называются *управляющими битами*, потому что они указывают процессору, какую операцию надо выполнить. Например, поля `op` и `funct` для инструкции `add` выглядят так: `op` = 51 (0110011_2), `funct7` = 0 (0000000_2) и `funct3` = 0 (000_2). Аналогично инструкция `sub` состоит из полей: `op` = 51, `funct7` = 32 (0100000_2) и `funct3` = 0 (000_2). На **рис. 6.13** показан машинный код для двух инструкций типа *R* – `add` и `sub`. Два регистра источника и регистр-назначение закодированы в трех полях: `rs1`, `rs2` и `rd`. Поля содержат номера регистров, приведенные в **табл. 6.1**. Например, `s0` – это регистр 8 (x8). Обратите внимание, что в инструкциях ассемблера и машинного языка регистры расположены во взаимно обратном порядке. Например, ассемблерная инструкция `add s2, s3, s4` имеет вид `rd = s2` (18), `rs1 = s3` (19) и `rs2 = s4` (20). Эти регистры перечислены слева направо в инструкции ассемблера, но следуют справа налево в машинном коде.

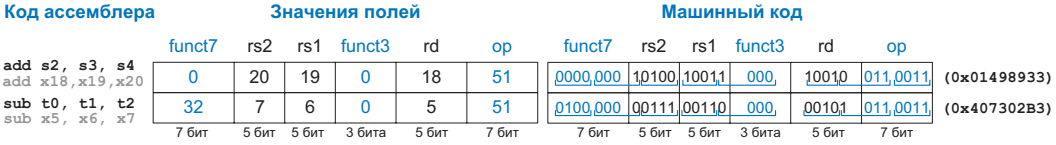


Рис. 6.13 Машинный код для инструкций типа *R*

В **табл. B.1 приложения B** перечислены коды операций и поля функций (`funct3` и `funct7`) для набора инструкций RV32I. Самый простой способ перевести инструкцию из ассемблера в машинный код (аналогичный показанному на **рис. 6.13**) – это записать значения каждого поля и преобразовать их в двоичный код. Затем следует сгруппировать биты в блоки по четыре, чтобы преобразовать их в шестнадцатеричные числа и сделать запись машинного языка более компактной.

К командам типа *R* относятся также команды сдвига (`sll`, `srl` и `sra`) и логические операции (`and`, `or`, и `xor`). Команды сдвига с непосредственным операндом (`slli`, `srli` и `srai`) представляют собой команды типа *I*, которые мы обсудим в **разделе 6.4.2**.

На **рис. 6.14** показан машинный код для операций логического сдвига влево (`sll`) и `xor`. Код операции 51 (0110011_2) одинаковый для всех операций типа *R*. Команды сдвига, у которых величина сдвига указана в регистре (`sll`, `srl` и `sra`), сдвигают значение `rs1` на 5-битное значение без знака в битах 4:0 регистра `rs2` и помещают результат в `rd`. Для всех команд сдвига в полях `funct7` и `funct3` закодирован тип сдвига или логической операции, которую необходимо выполнить, как показано в **табл. B.1**. Например, для команды `sll` применяются значения полей `funct7` = 0 и `funct3` = 1; для команды `xor` – значения `funct7` = 0 и `funct3` = 4.

Код ассемблера	Значения полей						Машинный код						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`sll s7, t0, s1`	0	9	5	1	23	51	0000 000	01001	00101	001	10111	011 0011	(0x00929BB3)
`xor s8, s9, s10`	0	26	25	4	24	51	0000 000	11010	11001	100	11000	011 0011	(0x01ACCC33)
	7 бит	5 бит	5 бит	3 бита	5 бит	7 бит	7 бит	5 бит	5 бит	3 бита	5 бит	7 бит	

Рис. 6.14 Еще один пример машинного кода для инструкций типа *R*

Инструкции типа *R* состоят из 17 бит кодов операций и функций, которых достаточно для представления 2^{17} = 131 072 различных инструкций. Это количество выглядит чрезмерным, учитывая, что до сих пор мы упоминали менее дюжины инструкций типа *R*. Соответственно, для кодирования регистров источника и назначения остается еще 15 бит. Это свободное пространство набора команд открывает перед архитектурой RISC-V большие перспективы для расширения. Например, расширение RISC-V F Extension добавляет инструкции с плавающей запятой, описанные далее в разделе 6.6.4 и в приложении B.

Пример 6.3 ПРЕОБРАЗОВАНИЕ ИНСТРУКЦИЙ ТИПА *R* ИЗ ПРЕДСТАВЛЕНИЯ НА ЯЗЫКЕ АССЕМБЛЕРА В МАШИННЫЙ КОД

Преобразуйте приведенную ниже ассемблерную инструкцию RISC-V в машинный код:

```
add t3, s4, s5
```

Решение Согласно **табл. 6.1**, номера регистров `t3`, `s4` и `s5` равны 28, 20 и 21 соответственно. Согласно **табл. B.1**, поле `opcode` для команды `add` равно 51 (01100112), поле `funct` разбито на значения `funct7` = 0 и `funct3` = 0. Эти поля и соответствующий машинный код показаны на **рис. 6.15**. Простейший способ получить шестнадцатеричный машинный код – это сначала записать машинный код в двоичном виде, а затем поделить его на группы по четыре бита (тетрады), которые заменить соот-

ветствующими шестнадцатеричными цифрами (показанными на рисунке синим цветом). Таким образом, для этой инструкции машинный код равен 0x015A0E33.

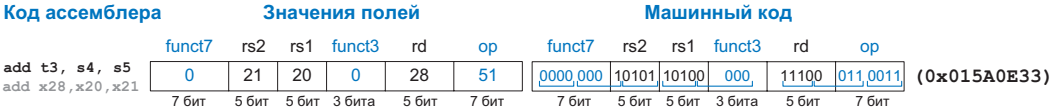


Рис. 6.15 Машинный код для инструкции типа *R* из примера 6.3

6.4.2. Инструкции типа I

Команды типа *I* (immediate, непосредственные) используют два регистровых операнда и один непосредственный операнд (константу). К инструкциям типа *I* относятся `addi`, `andi`, `ori` и `xori`, операции загрузки (`lw`, `lh`, `lb`, `lhu` и `lbu`) и регистрового перехода (`jalr`). На рис. 6.16 показан формат машинных команд типа *I*. Он похож на формат команд типа *R*, но вместо полей `funct7` и `rs2` содержит12-битное непосредственное поле `imm`. Поля `rs1` и `imm` представляют собой операнды-источники, а поле `rd` – регистр-назначение.

Рис. 6.16 Формат инструкции типа *I*

На рис. 6.17 приведено несколько примеров кодирования инструкций типа *I*. Поле константы представляет собой 12-битное число со знаком (в дополнительном коде) для всех инструкций типа *I*, кроме инструкций непосредственного сдвига `slli`, `srli` и `srai`. Для этих трех инструкций сдвига поле $imm_{4:0}$ представляет собой 5-битное значение сдвига без знака; верхние семь бит `imm` равны 0 для инструкций `srli` и `slli`, но инструкция `srai` помещает 1 в imm_{10} (т. е. 30-й бит инструкции), как показано на рис. 6.17. Как и в командах типа *R*, порядок операндов в командах ассемблера типа *I* отличается от такового в машинном представлении.

Рис. 6.17 Машинный код для инструкций типа *I*

Пример 6.4 ТРАНСЛЯЦИЯ ИНСТРУКЦИЙ ТИПА *I* В МАШИННЫЙ КОД

Преобразуйте приведенную ниже инструкцию в машинный код.

```
lw t3, -36 (s4)
```

Решение Согласно **табл. 6.1**, номера регистров `t3` и `s4` равны 28 и 20 соответственно. Поле `rs1` (`s4` = `x20`) указывает на регистр `s4`, содержащий базовый адрес, а поле `rd` (`t3` = `x28`) указывает на регистр-назначение `t3`. Поле `imm`, хранящее непосредственный операнд, содержит 12-битное смещение, равное −36. В **табл. B.1** мы видим, что инструкции `lw` соответствуют `op` = `3` (0000011_2) и `funct3` = 2 (010_2). Коды полей и получившийся машинный код показаны на **рис. 6.18**.

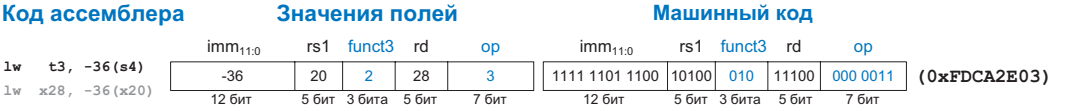


Рис. 6.18 **Машинный код для инструкции типа *I* из примера 6.4**

Напомним, что *M*-битное число в дополнительном коде дополняется знаком до *N*-битного числа ($N > M$) путем копирования знакового бита (самого старшего значимого бита) *M*-битного числа во все старшие биты *N*-битного числа. Эта операция не меняет исходное значение, представленное в дополнительном коде. Например, 1101_2 — это 4-битная запись в дополнительном коде, соответствующая -3_{10}. При расширении знаком до 8 бит двоичная запись приобретает вид 11111101_2, но по-прежнему соответствует -3_{10}.

Инструкции типа *I* содержат 12-битное поле константы, но константы участвуют в 32-битных операциях. Например, инструкция `lw` добавляет 12-битное смещение к 32-битному базовому адресу. Что же произойдет в верхних 20 битах из 32? У неотрицательных констант все старшие биты будут заполнены нулями, а у отрицательных констант они будут заполнены единицами. Напомним, что это называется дополнением знаковым битом.

6.4.3. Инструкции типа *S*/*B*

Аналогично инструкциям типа *I*, инструкции типа *S*/*B* (store/branch, хранение слова в памяти / условный переход) используют два регистровых операнда и один непосредственный операнд (константу). Но в инструкциях типа *S*/*B* оба операнда являются регистрами-источниками (`rs1` и `rs2`), тогда как инструкции типа *I* используют один регистр-источник (`rs1`) и один регистр-назначение (`rd`). На **рис. 6.19** показан формат машинного кода инструкций типа *S*/*B*. В отличие от инструкций типа *R*, здесь поля `funct7` и `rd` заменены на 12-битную константу `imm`. В коде машинного языка это поле константы разбивается на два битовых блока — биты 31:25 и биты 11:7.

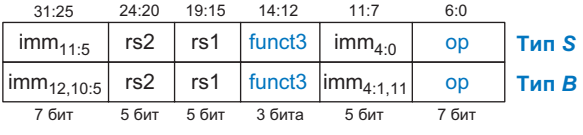


Рис. 6.19 **Форматы команд типа *S* и *B***

Инструкции сохранения слова в памяти используют тип *S*, а в инструкциях условного перехода ис-

пользуется тип *B*. Форматы *S* и *B* различаются только тем, как закодирована константа. Код инструкции типа *S* содержит 12-битную константу со знаком (в дополнительном коде), со старшими семью битами ($\texttt{imm}_{11:5}$) в битах 31:25 кода команды и младшими пятью битами ($\texttt{imm}_{4:0}$) в битах 11:7 кода команды.

Команды типа *B* содержат в коде 13-битовую константу со знаком, представляющую собой *смещение перехода* (branch offset), но только старшие 12 бит присутствуют в коде команды. Наименьший значащий бит всегда равен 0, потому что переход всегда представлен четным числом байтов; мы расскажем об этом немного позже. Константа в коде инструкции типа *B* выглядит несколько странно и слишком запутанно по сравнению с обычным представлением константы. Бит константы $\texttt{imm}_{12}$ находится в разряде машинного кода $\texttt{instr}_{31}$; $\texttt{imm}_{11}$ находится в $\texttt{instr}_{7}$; биты $\texttt{imm}_{10:5}$ находятся в $\texttt{instr}_{30:25}$; биты $\texttt{imm}_{4:1}$ находятся в $\texttt{instr}_{11:8}$; а $\texttt{imm}_{0}$ всегда равен нулю и, следовательно, вообще не является частью инструкции. Этот кажущийся битовый хаос устроили только для того, чтобы, насколько это возможно, биты константы занимали одни и те же биты кода команды в разных форматах и чтобы знаковый бит всегда находился в $\texttt{instr}_{31}$, как будет показано в **разделе 6.4.5**.

> В инструкции `sw` на языке ассемблера rs2 — это крайний левый регистр, то есть `sw rs2, offset(rs1)`.

На **рис. 6.20** показано несколько примеров кодирования инструкций сохранения с использованием формата типа *S*. Здесь поле `rs1` — базовый адрес, `imm` — смещение, а `rs2` — значение, которое будет сохранено в памяти. Напомним, что отрицательные константы представлены с помощью 12-битного значения в дополнительном коде. Например, в инструкции `sw x7, -6(x19)` регистр `x19` является базовым адресом (`rs1`), операнд `x7` (`rs2`) — значением, которое должно быть сохранено в памяти, а −6 — смещением. Для всех инструкций типа *S* значение `opcode` = 35 (01000112), а `funct3` может принимать значения `sb` (0), `sh` (1) и `sw` (2).

Код ассемблера	Значения полей						Машинный код						
	$\texttt{imm}_{11:5}$	rs2	rs1	funct3	$\texttt{imm}_{4:0}$	op	$\texttt{imm}_{11:5}$	rs2	rs1	funct3	$\texttt{imm}_{4:0}$	op	
`sw t2, -6(s3)` `sw x7, -6(x19)`	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
`sh s4, 23(t0)` `sh x20,23(x5)`	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
`sb t5, 0x2D(zero)` `sb x30,0x2D(x0)`	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
	7 бит	5 бит	5 бит	3 бита	5 бит	7 бит	7 бит	5 бит	5 бит	3 бита	5 бит	7 бит	

Рис. 6.20 Машинный код для инструкций типа *S*

Инструкции условного перехода `beq`, `bne`, `blt`, `bge`, `bltu` и `bgeu` используют формат типа *B*. На **рис. 6.21** показан пример кода с инструкцией перехода по условию «если равно» (`beq`). Адреса инструкций в памяти указаны слева от каждой инструкции. *Адрес перехода по условию* (branch target address, BTA) является конечной целью операции перехода. Инструкция `beq` на **рис. 6.21** содержит в своем коде BTA 0x80 −

это адрес метки L1. Смещение перехода дополняется знаковым битом и складывается с адресом текущей инструкции, образуя адрес перехода по условию.

```
#Адрес      # Ассемблер RISC-V
0x70        beq  s0, t5, L1   ⟩1
0x74        add  s1, s2, s3   ⟩2
0x78        sub  s5, s6, s7   ⟩3
0x7C        lw   t0, 0(s1)    ⟩4
0x80  L1:   addi s1, s1, -15
```

L1 находится на 4 инструкции (т. е. **16 байт**) после beq

$imm_{12:0}$ = 16	0		0	0	0	0	0	0	1	0	0	0	0
номер бита	12		11	10	9	8	7	6	5	4	3	2	1	0

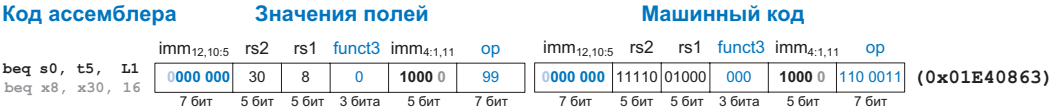


Рис. 6.21 Формат инструкций типа *S* и вычисления для инструкции beq

В инструкциях типа *B* операнды rs1 и rs2 являются двумя регистрами-источниками, а 13-битная константа смещения $imm_{12:0}$ определяет *количество байтов* между инструкцией перехода и BTA. В данном случае BTA – это четыре инструкции после инструкции beq, то есть $4 \times 4 = 16$ байт после beq. Следовательно, смещение перехода равно 16. В коде на машинном языке присутствуют только биты 12:1, поскольку бит 0 смещения перехода всегда равен нулю.

В машинном коде 32-битных инструкций биты 1:0 13-битного смещения перехода ($imm_{12:0}$) всегда равны нулю, поскольку 32-битные инструкции занимают 4 байта памяти. Поэтому адреса команд всегда делятся на четыре, и команда не нуждается в двух самых младших битах смещения перехода. Но машинный код набора инструкций RV32I отбрасывает только бит 0. Это обеспечивает совместимость с 16-битными (2-байтовыми) сжатыми инструкциями RISC-V (раздел 6.6.5). Компиляторы могут смешивать в одном коде на машинном языке 16- и 32-битные инструкции, если аппаратное обеспечение процессора поддерживает оба размера инструкций.

Пример 6.5 ТРАНСЛЯЦИЯ АССЕМБЛЕРНЫХ ИНСТРУКЦИЙ ТИПА B В КОД МАШИННОГО ЯЗЫКА

Рассмотрим демонстрационный фрагмент ассемблерного кода RISC-V. Адрес инструкции в памяти написан слева от каждой инструкции. Преобразуйте инструкцию перехода по условию «если не равно» (bne) в машинный код.

Адрес Инструкция

```
0x354     L1: addi s1, s1, 1
0x358         sub t0, t1, s7
...           ...
0xEB0         bne s8, s9, L1
```

Решение Согласно **табл. 6.1**, номерами регистров s8 и s9 являются 24 и 25 соответственно. Следовательно, rs1 = 24, а rs2 = 25. Метка L1 находится на 0xEB0 – 0x354 = 0xB5C (2908) байт перед инструкцией bne, значит, 13-битное значение смещения равно –2908 (1010010100100_2). Из **приложения B** следует, что opcode = 99 (1100011_2), а funct3 равно 1 (001_2). Получившийся машинный код показан на **рис. 6.22**. Обратите

внимание, что инструкции перехода могут переходить или вперед (к более высоким адресам) или, как в этом случае, назад (к более низким адресам).

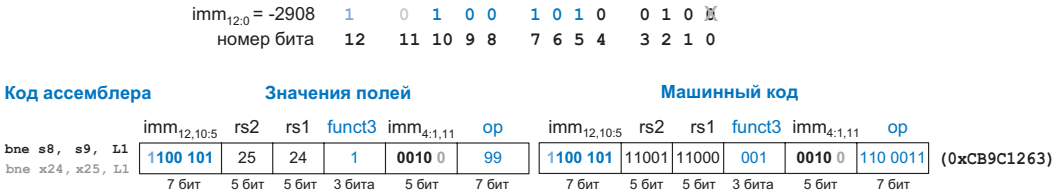


Рис. 6.22 Машинный код для инструкции типа *B* из примера 6.5

6.4.4. Инструкции типа *U*/*J*

Инструкции типа *U*/*J* (upper immediate/jump, старшие разряды константы / безусловный переход) содержат в своем машинном коде один операнд регистра-назначения `rd` и 20-битовое поле константы, как показано на **рис. 6.23**. Аналогично другим видам инструкций, инструкции типа *U*/*J* имеют 7-битный `opcode`. В инструкциях типа *U* оставшиеся биты отведены под 20 старших разрядов 32-битной константы. В инструкциях типа *J* оставшиеся биты отведены под 20 старших бит 21-битной константы смещения безусловного перехода. По аналогии с инструкциями типа *B*, самый младший значащий бит константы всегда равен 0 и не представлен в коде инструкции типа *J*.

Как и в случае с инструкциями типа *B*, биты констант в коде инструкций типа *J* странным образом перемешаны. Компьютерам все равно, но людей это раздражает.

На **рис. 6.24** показана переведенная в машинный код инструкция `lui`. Значение 32-битной константы состоит из 20 старших бит, закодированных непосредственно в инструкции, и нулей в младших битах. В данном случае после выполнения инструкции регистр `s5` (`rd`) содержит значение 0x8CDEF000.

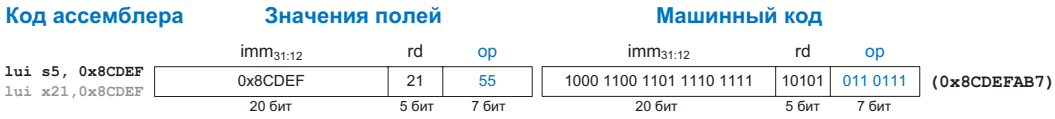


Рис. 6.24. Машинный код для инструкции `lui` типа *U*

На **рис. 6.25** показан пример кода с использованием инструкции `jal`. Адрес команды в памяти написан слева от каждой команды. Как и инструкции условного перехода, инструкции типа *J* выполняют переход к адресу, который основан на текущем состоянии программного счетчика и вычисляется при выполнении инструкции `jal`. На **рис. 6.25** *целевой адрес безусловного перехода* (jump target address, JTA) равен

Инструкция `jalr` относится к типу *I* (а не к типу *J*). Инструкция `jal` — это единственная инструкция типа *J*.

0xABC04, что на 0xA67F8 байт дальше после инструкции `jal`, расположенной по адресу 0x540C, потому что 0xABC04 – 0x540C = 0xA67F8 байт. Аналогично инструкции условного перехода, в коде инструкции младший бит отсутствует, потому что он всегда равен 0. Остальные биты присутствуют в 20-битовом поле константы, как показано на **рис. 6.25**. Если регистр-назначение `rd` не указан в ассемблерной инструкции `jal`, это поле по умолчанию имеет значение `ra` (`x1`). Например, инструкция `jal L1` эквивалентна инструкции `jal ra, L1`, и для нее `rd` = 1. Обычный безусловный переход `j` кодируется как инструкция `jal`, для которой `rd` = 0.

```
# Адрес              Ассемблер RISC-V
0x0000540C           jal ra, func1
0x00005410           add s1, s2, s3
...                  ...

0x000ABC04  func1:   add s4, s5, s8
...                  ...
```

`func1` равен **0xA67F8 байт** после `jal`

Рис. 6.25 **Машинный код для инструкции `jal` типа *J***

6.4.5. Кодирование констант

Формально архитектура RISC-V использует 32-битные константы со знаком, но фактически в коде команды умещаются только от 12 до 21 бита константы. На **рис. 6.26** показано, как формируются коды констант для каждого типа инструкций. Инструкции типа *I* и *S* используют 12-битные константы со знаком. В инструкциях типа *J* и *B* используются 21- и 13-битные константы со знаком, где младший бит всегда равен 0 (**разделы 6.4.3** и **6.4.4**). Инструкции типа *U* содержат в коде 20 старших бит 32-битной последовательности.

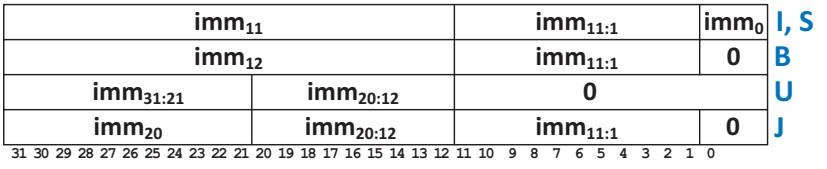


Рис. 6.26 **Константы в системе инструкций RISC-V**

Различия форматов команд в наборе RV32I стали следствием попытки сохранить расположение битов константы в одних и тех же битах команд с целью упрощения схемы на уровне аппаратуры (но за счет усложнения кодирования команд). На **рис. 6.27** показаны поля машинного кода для всех типов инструкций. (Код операции – это всегда биты 6:0, независимо от инструкции, поэтому он не показан на рисунке.) Поле $\texttt{instr}_{31}$ всегда хранит знаковый бит константы. Поле $\texttt{instr}_{30:20}$ содержит биты константы $\texttt{imm}_{30:20}$ для инструкций типа *U*. В ином случае поле $\texttt{instr}_{30:25}$ содержит биты константы $\texttt{imm}_{10:5}$. Поле $\texttt{instr}_{19:12}$ содержит биты константы $\texttt{imm}_{19:12}$ для инструкций типа *U*/*J*. Биты константы $\texttt{imm}_{4:1}$ занимают либо поле $\texttt{instr}_{24:21}$, либо поле $\texttt{instr}_{11:8}$. Бит константы 11 (когда это незнаковый бит) и бит 0 – это «блуждающие» биты, которые хранятся в бите 0 или 20 кода инструкции.

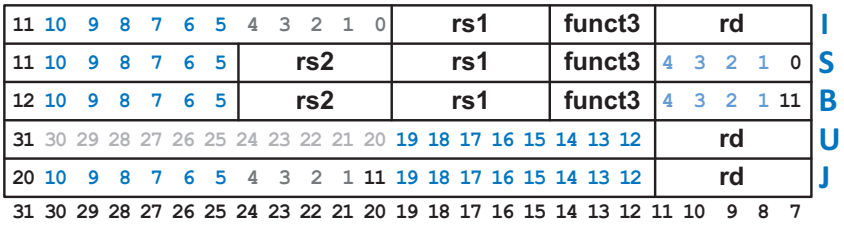


Рис. 6.27 Представление констант в машинном коде RISC-V

Поддержание единообразия расположения битов в разных форматах команд – еще один пример постоянства, упрощающего конструкцию; в частности, это сводит к минимуму количество соединений и мультиплексоров, необходимых для извлечения констант и дополнения их знаковым битом. В **упражнениях 6.47** и **6.48** показаны более глубокие последствия этого конструктивного решения для аппаратной реализации процессора.

6.4.6. Режимы адресации

От *режима адресации* зависит, каким образом определяются операнды, участвующие в инструкции. В этом разделе кратко описаны режимы, применяемые для адресации операндов команд. Архитектура RISC-V использует *четыре режима адресации*: регистровую (register-only), непосредственную (immediate), базовую (base) и относительно счетчика команд (PC-relative). Большинство других архитектур используют аналогичные режимы адресации, поэтому понимание упомянутых режимов поможет вам изучить и другие языки ассемблера. Первые три режима (регистровая, непосредственная и базовая адресация) определяют режимы чтения и записи операндов. Последний режим (относительно счетчика команд) определяет способ записи *счетчика команд* (program counter, PC).

Регистровая адресация

При *регистровой адресации* регистры используются для всех операндов-источников и операндов-назначений (иными словами – для всех операндов и результата). Все инструкции типа R применяют именно такой режим адресации.

Непосредственная адресация

При *непосредственной адресации* в качестве операндов наряду с регистрами используют константы (непосредственные операнды). Некоторые инструкции типа *I*, такие как сложение с константой `addi` и логическая операция `xori`, применяют непосредственную адресацию с 12-битной константой со знаком. Команды сдвига с константой, определяющей величину сдвига `slli`, `srli` и `srai`, представляют собой инструкции типа *I*, которые помещают 5-битную величину непосредственного сдвига без знака в разряды $\texttt{imm}_{4:0}$ машинного кода. Формат команд загрузки `lb`, `lh` и `lw` аналогичен формату инструкций типа *I*, но при этом они используют базовую адресацию, которая обсуждается далее.

Базовая адресация

Инструкции доступа к памяти, такие как загрузка слова `lw` и сохранение слова `sw`, используют *базовую адресацию*. Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре `rs1` и 12-битного смещения с расширенным знаком, являющегося непосредственным операндом. Операции загрузки – это инструкции типа *I*, а операции сохранения – инструкции типа *S*.

Адресация относительно счетчика команд

Инструкции условного перехода используют *адресацию относительно счетчика команд* для определения нового значения счетчика команд в том случае, если нужно осуществить переход. Смещение со знаком, закодированное в поле константы, прибавляется к счетчику команд для определения нового значения PC; поэтому тот адрес, куда будет осуществлен переход, называют адресом *относительно* счетчика команд. Инструкции перехода по условию и `jal` используют для смещения 13- и 21-битные константы со знаком соответственно. Самые старшие значимые биты смещения располагаются

Инструкция безусловного перехода с возвратом (jalr) использует базовую адресацию, а не адресацию относительно PC. Она может выполнить переход к любому адресу инструкции в 32-битном адресном пространстве, потому что целевой адрес формируется сложением значения в `rs1` и 12-битной константы со знаком. Адрес возврата PC + 4 записывается в регистр-назначение.

Приведенная ниже последовательность инструкций позволяет программе перейти по любому адресу. Адреса инструкций указаны слева от каждой инструкции. В данном случае программа переходит к адресу 0x12345678 и записывает 0x0100FE7C (т. е. PC + 4) в регистр `t1`.

```
# Адрес     Ассемблер RISC-V
0x0100FE74 lui  s1, 0x12345
0x0100FE78 jalr t1, s1, 0x678
...        ...
0x12345678 ...
```

в 12- и 20-битных полях инструкций типа *B* и *J*. Наименьший значащий бит смещения всегда равен 0, поэтому он отсутствует в инструкции. Инструкция `auipc` (сложить старшие разряды константы смещения с PC) также использует адресацию относительно счетчика команд. Например, инструкция `auipc s3, 0xABCDE` помещает значение PC + 0xABCDE000 в регистр `s3`.

6.4.7. Расшифровываем машинные коды

Чтобы понимать машинный язык, нужно уметь расшифровывать поля каждой 32-битной команды. Для разных команд определены разные форматы, но во всех форматах команды начинаются с 7-битного поля `opcode`. Следовательно, чтение машинного кода нужно начинать с кода операции, чтобы определить, к какому типу принадлежит инструкция – *R*, *I*, *S* / *B* или *U* / *J*.

Пример 6.6 ТРАНСЛЯЦИЯ МАШИННЫХ КОДОВ НА ЯЗЫК АССЕМБЛЕРА

Преобразуйте приведенные ниже машинные коды на язык ассемблера.

```
0x41FE83B3
0xFDA48293
```

Решение Сначала мы записываем каждую инструкцию в двоичном коде и смотрим на семь младших бит, чтобы выяснить код операции (`opcode`) для каждой инструкции.

```
0100 0001 1111 1110 1000 0011 1011 0011 (0x41FE83B3)
1111 1101 1010 0100 1000 0010 1001 0011 (0xFDA48293)
```

От найденного кода операций зависит, как интерпретировать остальные биты. Код операции первой инструкции – 0110011_2; согласно **табл. B.1** в **приложении B**, это инструкция типа *R*, и мы можем разделить остальные биты на поля, соответствующие формату *R*, как показано в верхней части **рис. 6.28**. Код операции второй инструкции – 00100112, что соответствует инструкции *I* типа. Мы группируем оставшиеся биты согласно формату *I*, как показано в нижней части **рис. 6.28**. На этом рисунке представлен ассемблерный код, эквивалентный двум машинным инструкциям.

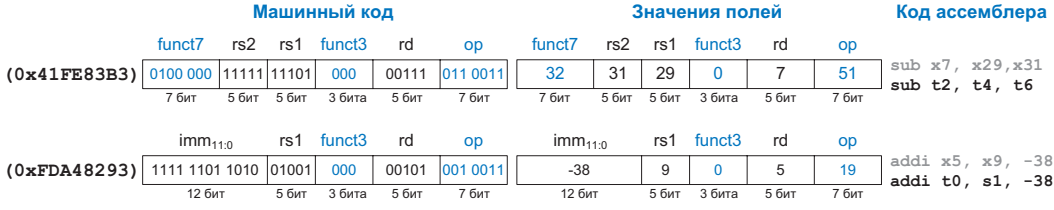


Рис. 6.28 **Трансляция машинного кода в код на языке ассемблера**

Ада Лавлейс, 1815–1852

Написала первую компьютерную программу. Программа предназначалась для вычисления чисел Бернулли на аналитической машине Чарльза Бэббиджа. Была единственным законнорожденным ребенком поэта лорда Байрона.

6.4.8. Могущество хранимой программы

Программа, записанная на машинном языке, – это последовательность чисел (в архитектуре RISC-V – 32-битных чисел), представляющих инструкции. Как и любые другие двоичные числа, эти инструкции можно хранить в памяти. Этот подход называется концепцией *хранимой программы* (stored program concept), и в нем заключается главная причина могущества компьютеров. Запуск новой программы не требует больших затрат времени и усилий на изменение или реконфигурацию аппаратного обеспечения; все, что для этого необходимо, – записать новую программу в память. Хранимые программы, в отличие от жестко зафиксированного аппаратного обеспечения, выполняющего лишь строго определенные функции, позволяют осуществлять *вычисления общего назначения* (general purpose computing). Используя этот подход, компьютер может выполнять любые приложения, начиная от простого калькулятора и заканчивая текстовыми процессорами и проигрывателями видео, просто меняя хранимую программу.

В хранимой программе команды считываются или выбираются (fetch) из памяти и выполняются процессором. Даже большие и сложные программы превращаются в последовательность операций чтения из памяти и выполнения команд. На **рис. 6.29** показано, как машинные инструкции хранятся в памяти. В программах для RISC-V инструкции обычно хранятся начиная с младших адресов, но это может зависеть от реализации. На **рис. 6.29** показан код, хранящийся между адресами 0x00000830 и 0x0000083C. Помните, что адресация памяти в архитектуре RISC-V побайтовая, поэтому адреса инструкций кратны четырем байтам, а не одному.

Чтобы запустить, или *выполнить*, хранимую программу, процессор последовательно выбирает ее команды из памяти. Далее выбранные команды расшифровываются (*дешифруются*) и выполняются аппаратным обеспечением. Адрес текущей команды хранится в 32-битном регистре, который называют *счетчиком команд* (program counter, PC).

Для того чтобы выполнить код, показанный на **рис. 6.29**, операционная система загружает в счетчик команд значение 0x00000830. Процессор читает из памяти по этому адресу команду 0x01498933 (`add s2, s3, s4`) и выполняет ее. Затем процессор увеличивает значение счетчика команд на 4 (оно становится равным 0x00000834), выбирает из памяти и выполняет новую команду, после чего процесс повторяется.

Код ассемблера	Машинный код
`add  s2, s3, s4`	`0x01498933`
`sub  t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw   t2, -6(s3)`	`0xFFA9A383`

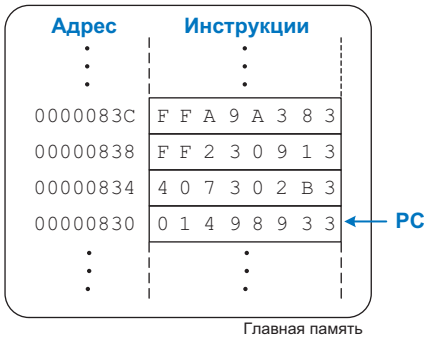


Рис. 6.29 Хранимая программа

Архитектурное состояние (architectural state) микропроцессора описывает состояние программы. Архитектурное состояние процессоров RISC-V включает в себя содержимое памяти, регистрового файла и счетчика команд. Если операционная система в какой-либо момент выполнения программы сохранит архитектурное состояние, то сможет эту программу прервать, сделать что-то еще, а потом восстановить архитектурное состояние, после чего прерванная программа продолжит выполняться, даже не узнав, что ее вообще прерывали. Архитектурное состояние будет играть важную роль, когда мы приступим к созданию микропроцессора в **главе 7**.

6.5. Камера, мотор! Компилируем, ассемблируем и загружаем

Ранее мы показали, как небольшие фрагменты кода, написанного на языке высокого уровня, транслируются в ассемблерный и машинный коды. В этом разделе мы рассмотрим, как происходит компиляция и ассемблирование целой программы, написанной на языке высокого уровня, и покажем, как загрузить ее в память компьютера для выполнения. Мы начнем с рассмотрения карты памяти RISC-V, описывающей расположение кода, данных и стека в памяти.

На **рис. 6.30** показаны этапы, необходимые для трансляции в машинный язык и начала выполнения программы, разработанной на языке вы-

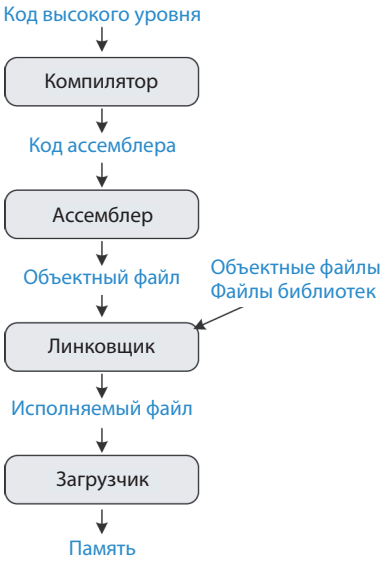


Рис. 6.30 Этапы трансляции и запуска программы

Рис. 6.31 Пример карты памяти RISC-V

сокого уровня. Высокоуровневый код компилируется в код на языке ассемблера, который затем ассемблируется в машинный код и сохраняется в виде объектного файла. *Компоновщик*, также называемый *редактором связей*, или *линкером* (linker), объединяет полученный объектный код с объектным кодом библиотек и других файлов, в результате чего получается готовая к исполнению программа. На практике большинство компиляторных пакетов выполняют все три шага: компиляцию, ассемблирование и компоновку. Наконец, загрузчик загружает программу в память и запускает ее. В оставшейся части этого раздела мы более подробно рассмотрим эти этапы на примере простой программы.

6.5.1. Карта памяти

Так как архитектура RISC-V использует 32-битные адреса, то размер адресного пространства составляет 2^{32} байта = 4 гигабайта (Гбайта). Адреса слов кратны 4 и располагаются в промежутке от 0 до 0xFFFFFFFC. На **рис. 6.31** показан пример карты памяти. Адресное пространство разделено на пять частей, или *сегментов*: сегмент кода и постоянных данных (text segment), сегмент глобальных данных, сегмент динамических данных, а также сегмент для обработчиков исключений и сегмент операционной системы (ОС), который включает в себя отображение ввода/вывода на пространство памяти (I/O segment). Эти сегменты рассматриваются в следующих разделах. Мы представляем здесь условный пример карты памяти RISC-V, поскольку спецификация архитектуры RISC-V не определяет конкретную карту памяти. Согласно существующему соглашению, обработчик исключений обычно располагается либо по нижнему, либо по верхнему адресу, а пользователь по своему усмотрению определяет, где будет размещаться сегмент кода и постоянных данных, отображаемый в память ввод-вывод, стек и глобальные данные. Это обеспечивает гибкость, особенно при разработке небольших систем, таких как карманные устройства, где используется только часть диапазона памяти и, как следствие, можно обойтись небольшой физической памятью.

Сегмент кода

В *сегменте кода* хранится пользовательская программа на машинном языке. Помимо кода, он может включать литералы (константы) и данные только для чтения.

Сегмент глобальных данных

Сегмент глобальных данных (global data segment) содержит глобальные переменные, которые, в отличие от локальных переменных, находятся в области видимости всех функций в программе. Локальные переменные определяются внутри функции и могут быть видны только этой функции; они обычно находятся в регистрах или в стеке. Глобальные переменные размещаются в памяти до начала выполнения программы, и обычно к ним обращаются с помощью регистра *глобального указателя* `gp` (регистр x3), который в начале выполнения программы содержит адрес середины сегмента глобальных данных `gp` = 0x10000800. Во время ассемблирования смещение уже известно, так что, используя 12-битное смещение со знаком, программисты могут получить доступ ко всему глобальному сегменту данных.

Архитектура RISC-V требует, чтобы указатель стека `sp` поддерживал 16-байтовое выравнивание для обеспечения совместимости с базовым набором команд RISC-V с четырехкратной точностью RV128I, который работает со 128-битными (т. е. 16-байтовыми) данными. Таким образом, при выделении места в стеке значение `sp` уменьшается на величину, кратную 16, даже если требуется меньшее количество места в стеке. Мы умолчали об этом требовании в разделе 6.3.7, чтобы не отвлекать внимания от описания основных функций стека.

Сегмент динамических данных

Сегмент динамических данных содержит стек и кучу. В момент запуска программы этот сегмент не содержит данных – они динамически выделяются и освобождаются в нем в процессе выполнения программы.

При запуске операционная система устанавливает указатель стека (`sp`, регистр `x2`) так, чтобы он указывал на вершину стека. Стек растет вниз от верхней границы сегмента динамических данных (`sp` = 0xBFFFFFF0), а доступ к фреймам стека осуществляется в режиме очереди LIFO («последним пришел – первым ушел»).

Куча (heap) хранит блоки памяти, динамически выделяемые программе во время работы. В языке C выделение памяти осуществляется функцией `malloc`; в C++ и Java для этого служит функция `new`. Как и в случае кучи одежды на полу комнаты в общежитии, данные, находящиеся в куче, можно использовать и выбрасывать в произвольном порядке.

Куча растет вверх от нижней границы сегмента динамических данных. Если стек и куча прорастут друг в друга, данные программы могут быть повреждены. Функция выделения памяти стремится избежать этой ситуации. Она возвращает *ошибку нехватки памяти* (out-of-memory error), если свободной памяти недостаточно для размещения новых динамических данных.

Обработчик исключений, ОС и сегменты ввода-вывода (I/O)

Самая нижняя часть приведенной в качестве примера карты памяти RISC-V зарезервирована для *обработчиков исключений* (**раздел 6.6.2**) и загрузочного кода, который запускается при запуске. Самая верхняя часть карты памяти зарезервирована для операционной системы и отображения ввода/вывода (I/O) на пространство памяти (**раздел 9.2**).

6.5.2. Директивы ассемблера

Директивы ассемблера помогают ассемблеру выделять и инициализировать глобальные переменные, определять константы и различать код и данные. В **табл. 6.5** перечислены наиболее часто применяемые директивы ассемблера RISC-V, а в **примере кода 6.29** показано, как их использовать.

Таблица 6.5 Директивы ассемблера RISC-V

Директива ассемблера	Описание
`.text`	Секция кода (text section)
`.data`	Секция глобальных данных
`.bss`	Глобальные данные, инициализированные нулями
`.section .foo`	Секция с именем `.foo`
`.align N`	Выравнивание последующих данных/команд по границе, кратной 2^N
`.balign N`	Выравнивание последующих данных/команд по границе, кратной *N*
`.globl sym`	Метка `sym` объявлена глобальной
`.string "str"`	Сохранение строки «str» в памяти
`.word w1, w2, ..., wN`	Сохранение *N* 32-битовых переменных в последовательных словах памяти
`.byte b1, b2, ..., bN`	Сохранение *N* 8-битовых переменных в последовательных словах памяти
`.space N`	Резервирует *N* байт под хранение переменных
`.equ name, constant`	Определяет символьное имя `name` для значения `constant`
`.end`	Конец ассемблерного кода

Руководствуясь директивами `.data`, `.text`, `.bss` и `.section .rodata`, ассемблер размещает в памяти обрабатываемые данные или код в сегментах глобальных данных, текст (код), BSS или данные толь-

ко для чтения (.rodata) соответственно. Сегмент BSS находится в сегменте глобальных данных, но инициализируется нулями. Сегмент данных только для чтения – это константы, которые помещаются в сегмент кода (т. е. в *память программы*).

Программа в **примере кода 6.29** начинается с того, что метку main объявляют глобальной (.globl main), и теперь функцию main можно вызывать извне этого кода. Обычно это делают ОС или загрузчик. Затем значение N устанавливают равным 5 (.equ N,5). Ассемблер заменяет N на 5 перед трансляцией инструкций ассемблера в машинный код. Например, инструкция lw t5, N*4 (t0) преобразуется в lw t5, 20 (t0), а затем транслируется в машинный код (0x0142AF03). Затем программа выделяет следующие глобальные переменные, как показано на **рис. 6.32**: A (массив 32-байтовых значений из 7 элементов), str1 (строка с нулевым символом в конце), B и C (по 4 байта каждая) и D (1 байт). Переменные A и B и строка str1 инициализируются соответственно значениями {5, 42, −88, 2, −5033, 720, 314}, 0x32A и «RISC-V» (т. е. {52, 49, 53, 43, 2D, 56, 00} согласно **табл. 6.2**). Помните, что в языке программирования C строки заканчиваются нулевым символом (0x00). Переменные C и D не инициализированы пользователем и находятся в сегменте BSS. Компилятор включил по 16 байт нераспределенной памяти между сегментами данных и BSS, как показано серыми полями на **рис. 6.32**.

Директива ассемблера .align 2 выравнивает данные или код по границе $2^2 = 4$ байта. Ей эквивалентна директива ассемблера .balign 4 (byte align 4). Эти директивы ассемблера помогают поддерживать целостность компоновки данных и инструкций. Например, если бы мы не поставили директиву .align 2 перед выделением памяти переменной B (т. е. перед B: .word 0x32A), то место под B было бы выделено непосредственно после переменной str1 в байтах 0x2157–0x215A (вместо 0x2158–0x215B).

Функция main начинается с загрузки адресов глобальных переменных в t0–t4 с помощью псевдоинструкции загрузки адреса la (**табл. В.7** в **приложении В**). Программа извлекает значения A[5] и C из памяти, складывает их и помещает результат (0x5FA) в D. Затем она загружает из памяти значение str1[4] (которое равно '-' = код ASCII 0x2D), используя инструкцию lb t5, N–1 (t1), и размещает это значение в глобальной переменной B. В конце программа читает значе-

Сокращение BSS означает *символ начала блока* (block started symbol), и изначально это было ключевое слово для выделения блока неинициализированных данных. Теперь большинство операционных систем инициализируют данные в сегменте BSS нулями.

Программа из **примера кода 6.29** была запущена на коммерческом ядре SweRV EH1 RISC-V Western Digital с открытым исходным кодом. Другие процессоры используют иные карты памяти, поэтому переменные и код в них могут быть размещены по другим адресам. Пройдя бесплатный курс RVfpga (RISC-V FPGA) от Imagination Technologies, вы узнаете, как использовать ядро SweRV EH1, предназначенное для FPGA, для запуска программ на языках C и ассемблера, а также изучите способы расширения и модификации этого процессора и системы команд RISC-V. Больше подробностей по ссылке: https://univesity.imgtec.com/rvfpga/.

Обратите внимание, что строка str2 находится в сегменте кода (а не в сегменте данных) по адресу 0x140, рядом с кодом пользовательской программы (main), который начинается с адреса 0x88. Объединяя код и данные, можно минимизировать объем необходимой памяти и количество инструкций для доступа к данным, которые имеют решающее значение как для портативных, так и для встраиваемых систем.

ние `str2[8]`, которое является символом `"r"`, и помещает это значение в `str1[0]`. Функция `main` завершается возвратом к операционной системе или загрузочному коду с помощью инструкции `jr ra`. На **рис. 6.33** показаны значения `C`, `D` и `str1` после завершения программы. Директива ассемблера `.end` указывает на конец ассемблерного кода.

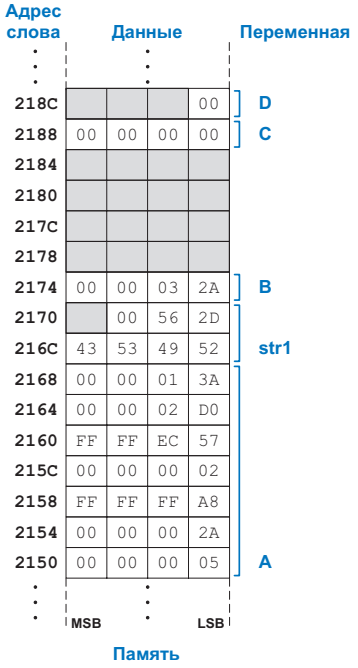


Рис. 6.32 Распределение памяти глобальных переменных в примере кода 6.29

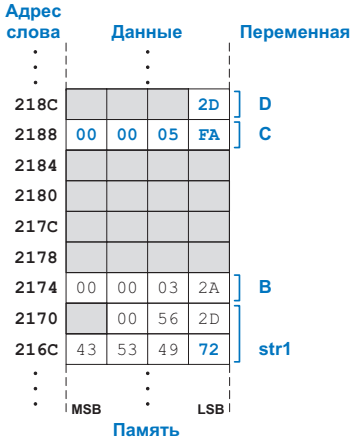


Рис. 6.33 Конечные значения глобальных переменных `C`, `D` и `str1`

Пример кода 6.30 ИСПОЛЬЗОВАНИЕ ДИРЕКТИВ АССЕМБЛЕРА

```
.globl main          # делает метку main глобальной
.equ N, 5            # N = 5
.data                # сегмент глобальных данных
A: .word 5, 42, -88, 2, -5033, 720, 314
str1: .string "RISC-V"
.align 2             # выравнивает следующие данные по 2^2-байтовой границе
B: .word 0x32A
.bss                 # сегмент bss - переменные инициализированы нулями
C: .space 4
D: .space 1
.balign 4            # выравнивает следующие команды по 4-байтовой границе
.text                # сегмент кода
main:
la  t0, A            # t0 = адрес A = 0x2150
```

Пример кода 6.30 ИСПОЛЬЗОВАНИЕ ДИРЕКТИВ АССЕМБЛЕРА

```
la  t1, str1        # t1 = адрес str1 = 0x216C
la  t2, B           # t2 = адрес B = 0x2174
la  t3, C           # t3 = адрес C = 0x2188
la  t4, D           # t4 = адрес D = 0x218C
lw  t5, N*4(t0)     # t5 = A[N] = A[5] = 720 = 0x2D0
lw  t6, 0(t2)       # t6 = B = 810 = 0x32A
add t5, t5, t6      # t5 = A[N] + C = 720 + 810 = 1530 = 0x5FA
sw  t5, 0(t3)       # C = 1530 = 0x5FA
lb  t5, N-1(t1)     # t5 = str1[N-1] = str1[4] = ,-' = 0x2D
sb  t5, 0(t4)       # D = str1[N-1] = 0x2D
la  t5, str2        # t5 = адрес str2 = 0x140
lb  t6, 8(t5)       # t6 = str2[8] = ,r' = 0x72
sb  t6, 0(t1)       # str1[0] = ,r' = 0x72
jr  ra              # возврат
.section .rodata
str2: .string "Hello world!"
.end # конец ассемблерного файла
```

6.5.3. Компиляция

Компилятор транслирует код высокого уровня на язык ассемблера, а затем ассемблер транслирует его в машинный код. Примеры в этом разделе основаны на использовании GCC, популярного и широко используемого бесплатного компилятора. GCC является частью набора инструментов, который предлагает и другие возможности. Некоторые из них мы обсудим в этом разделе. В **примере кода 6.30** показана простая высокоуровневая программа с тремя глобальными переменными и двумя функциями, а также ассемблерный код, созданный компилятором GCC, входящим в набор инструментов SiFive Freedom E SDK. Об использовании компиляторов RISC-V подробнее сказано в **предисловии**.

В **примере кода 6.30** функция `main` начинается с сохранения `ra` в стеке. Она оставляет место для четырех слов (16 байт), но использует только одно из них. Напомним, что указатель стека `sp` должен поддерживать 16-байтовое выравнивание для совместимости с системой инструкций RV128I. Затем функция main записывает значение 2 в глобальную переменную `f` и 3 в глобальную переменную `g`. Глобальные переменные пока не размещены в памяти – это сделает ассемблер. Обратите внимание, что в данном примере ассемблерный код использует две инструкции (`lui`, за которой следует `sw`) вместо

Грейс Хоппер, 1906–1992

Окончила Йельский университет со степенью доктора философии по математике (англ. Ph. D., западный аналог степени кандидата математических наук). Во время работы на корпорацию Remington Rand разработала первый компилятор. Сыграла важную роль в разработке языка программирования COBOL. Будучи офицером ВМФ, получила множество наград, в том числе медаль за победу во Второй мировой войне и медаль за службу национальной обороне. Она также задокументировала первый в истории компьютерный «баг» (bug, жучок), который в данном случае был настоящим насекомым, прилипшим к перфокарте.

одной (`sw`) для сохранения каждой глобальной переменной, поскольку необходимо указывать 32-битный адрес.

Пример кода 6.30 КОМПИЛЯЦИЯ ПРОГРАММЫ ВЫСОКОГО УРОВНЯ

Код на языке высокого уровня

```
int f, g, y;


int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b - 1));
}




                                  void main() {
                                    f=2;
                                    g=3;
                                    y=func(f,g);

                                    return;
                                  }
```

Код на языке ассемблера RISC-V

```
      .text
      .globl        func
      .type func,   @function
func:
      addi  sp,sp,-16
      sw    ra,12(sp)
      sw    s0,8(sp)
      mv    s0,a0
      add   a0,a1,a0
      bge   a1,zero,.L5
.L1:
      lw    ra,12(sp)
      lw    s0,8(sp)
      addi  sp,sp,16
      jr    ra
.L5:
      addi  a1,a1,-1
      mv    a0,s0
      call  func
      add   a0,a0,s0
      j     .L1

     .globl         main
     .type main,    @function
main:
      addi  sp,sp,-16
      sw    ra,12(sp)
      lui   a5,%hi(f)
      li    a4,2
      sw    a4,%lo(f)(a5)
      lui   a5,%hi(g)
      li    a4,3
      sw    a4,%lo(g)(a5)
      li    a1,3
      li    a0,2
      call  func
      lui   a5,%hi(y)
      sw    a0,%lo(y)(a5)
      lw    ra,12(sp)
      addi  sp,sp,16
      jr    ra
      .comm y,4,4
      .comm g,4,4
      .comm f,4,4
```

Затем программа помещает `f` и `g` (т. е. 2 и 3) в регистры аргументов `a0` и `a1` и вызывает функцию `func` при помощи псевдоинструкции `call`

func. Функция func сохраняет значения ra и s0 в стеке. Потом она сохраняет значение a0 (a) в s0 (потому что оно понадобится после рекурсивного вызова func) и вычисляет a0 = a0 + a1 (возвращаемое значение = a + b). Затем функция func выполняет переход, если значение a1 (b) больше или равно нулю. В противном случае она восстанавливает значения ra, s0 и sp и возвращается из вызова при помощи jr ra. Если же произошел переход по условию b ≥ 0, то функция func уменьшает на единицу значение a1 (b) и рекурсивно вызывает сама себя. Вернувшись из рекурсивного вызова, она складывает возвращаемое значение a0 с s0 (a) и переходит к метке .L1, где восстанавливает значения ra, s0 и sp и выполняет возврат из вызова. Затем функция main сохраняет результат, возвращенный функцией func (a0), в глобальную переменную y, восстанавливает ra и sp и возвращает y. В нижней части ассемблерного кода программа указывает, что у нее есть три глобальные переменные шириной 4 байта – y, g и f, – используя директиву .comm y, 4, 4 и т. д. Первая четверка обозначает 4-байтовое выравнивание, а вторая четверка указывает размер переменной (4 байта).

Теперь давайте выполним компиляцию, трансляцию и компоновку программы на языке C с именем prog.c при помощи компилятора GCC. Для этого наберите в окне терминала команду

```
gcc -O1 -g prog.c -o prog
```

Эта команда создает исполняемый выходной файл с именем prog. Флаг -O1 просит компилятор выполнить базовую оптимизацию, а не производить крайне неэффективный код. Флаг -g указывает компилятору включить отладочную информацию в файл.

Чтобы наблюдать промежуточные шаги, воспользуйтесь флагом GCC -S, и тогда после компиляции не будут выполняться шаги сборки и компоновки:

```
gcc -O1 -S prog.c -o prog.s
```

Компилятор выводит в файл prog.s довольно подробную информацию. Чтобы не перегружать ваше внимание, мы показали наиболее интересную часть вывода в **примере кода 6.30**.

6.5.4. Трансляция

Ассемблер транслирует код на языке ассемблера в объектный файл, содержащий код на машинном языке. Воспользуйтесь следующими командами, чтобы создать объектный файл либо из prog.s, либо непосредственно из prog.c:

```
gcc -c prog.s -o prog.o
```

или

```
gcc -O1 -g -c prog.c -o prog.o
```

Ассемблер выполняет два прохода по ассемблерному коду. Во время первого прохода ассемблер назначает командам адреса и находит все символы, такие как метки и имена глобальных переменных. Имена и адреса символов хранятся в *таблице символов*. Во время второго прохода ассемблер генерирует машинный код. Адреса глобальных переменных и меток берутся из таблицы символов. Код на машинном языке и таблица символов сохраняются в объектном файле.

Мы можем *дизассемблировать* объектный файл с помощью команды `objdump`, чтобы увидеть код языка ассемблера рядом с кодом машинного языка:

```
objdump -S prog.o
```

Ниже показан результат дизассемблирования раздела `.text`. Если код был изначально скомпилирован с флагом `-g`, то дизассемблер также покажет соответствующие строки кода C, сопровождая их вкраплениями ассемблерного кода. Обратите внимание, что псевдоинструкция `call` была транслирована в две инструкции RISC-V: `auipc ra, 0x0` и `jalr ra`. В данном случае вызываемая функция находится далеко, то есть разница адреса перехода и текущего значения PC больше, чем можно достигнуть прибавлением 21-битного смещения со знаком в инструкции `jal`. Инструкции сохранения значений в глобальные переменные на данном этапе являются просто *заполнителями* (placeholders) до тех пор, пока глобальные переменные не размещены в памяти. Например, три инструкции по адресам от 0x48 до 0x50 предназначены для сохранения значения 2 в глобальной переменной `f`. Как только на этапе компоновки переменная `f` получит свое место в памяти, инструкции будут обновлены.

```
00000000 <func>:
int f, g, y;
int func(int a, int b) {
   0: ff010113                addi sp,sp,-16
   4: 00112623                sw ra,12(sp)
   8: 00812423                sw s0,8(sp)
   c: 00050413                mv s0,a0
   if (b<0) return (a+b);
  10: 00a58533                add a0,a1,a0
  14: 0005da63                bgez a1,28 <.L5>
00000018 <.L1>:
  else return(a + func(a, b-1));
}
  18: 00c12083                lw ra,12(sp)
  1c: 00812403                lw s0,8(sp)
  20: 01010113                addi sp,sp,16
  24: 00008067                ret
00000028 <.L5>:
  else return(a + func(a, b-1));
  28: fff58593                addi a1,a1,-1
  2c: 00040513                mv a0,s0
```

```
  30: 00000097                auipc ra,0x0
  34: 000080e7                jalr ra # 30 <.LVL5+0x4>
  38: 00850533 add a0,a0,s0
  3c: fddff06f                j 18 <.L1>
00000040 <main>:
void main() {
  40: ff010113                addi sp,sp,-16
  44: 00112623                sw ra,12(sp)
  f=2;
  48: 000007b7                lui a5,0x0
  4c: 00200713                li a4,2
  50: 00e7a023                sw a4,0(a5) # 0 <func>
  g=3;
  54: 000007b7                lui a5,0x0
  58: 00300713                li a4,3
  5c: 00e7a023                sw a4,0(a5) # 0 <func>
  y=func(f,g);
  60: 00300593                li a1,3
  64: 00200513                li a0,2
  68: 00000097                auipc ra,0x0
  6c: 000080e7                jalr ra # 68 <main+0x28>
  70: 000007b7                lui a5,0x0
  74: 00a7a023                sw a0,0(a5) # 0 <func>
  return;
}
  78: 00c12083                lw ra,12(sp)
  7c: 01010113                addi sp,sp,16
  80: 00008067                ret
```

Мы можем просмотреть таблицу символов из объектного файла, используя команду `objdump` с флагом `-t`. Результат выполнения команды показан ниже, мы лишь добавили названия для трех интересующих нас столбцов: адреса в памяти, размера и имени символа. Поскольку программа еще не помещена в память (не скомпонована), адреса пока являются только заполнителями. Символ `.Text` указывает на сегмент кода, а символ `.data` – на сегмент данных (глобальные данные). Размер этих двух символов в настоящее время равен 0, потому что программа еще не скомпонована. Размер двух функций `func` и `main` уже указан: `func` – это 0x40 (64) байт = 16 инструкций, а `main` – 0x44 (68) байт = 17 инструкций, как показано в приведенном выше коде. Здесь также перечислены символы глобальных переменных `f`, `g` и `y` по 4 байта каждый, но вместо адресов указано значение-заполнитель 0x00000004, поскольку им еще не назначены адреса в памяти.

Нас мало интересуют неподписанные столбцы в этой таблице символов. Они показывают флаги, связанные с символами (`l` для локальных или `g` для глобальных данных, `d` для отладки (debug), `F` для функции или `O` для объекта), и сегмент, в котором расположен символ (`.text`, `.data` или `*COM*` (common, общий), когда он не находится в разделе).

```
objdump -t prog.o
```

Адрес	Размер	Имя символа
00000000 l d .text	00000000	.text
00000000 l d .data	00000000	.data
00000000 g F .text	00000040	func

```
00000040 g F .text  00000044  main
00000004   O *COM*  00000004  f
00000004   O *COM*  00000004  g
00000004   O *COM*  00000004  y
```

6.5.5. Компоновка

Большие программы обычно содержат много файлов. Если программист изменяет один из этих файлов, то перекомпилировать и заново транслировать все остальные файлы выходит довольно затратно. Например, программы часто вызывают функции из библиотечных файлов, которые почти никогда не меняются, а соответствующие объектные файлы не нуждаются в обновлении. Кроме того, программа обычно включает в себя код запуска (для инициализации стека, кучи и т. д.), который должен быть выполнен перед вызовом основной функции.

Работа компоновщика заключается в том, чтобы объединить все объектные файлы в один-единственный файл с машинным кодом, который называется *исполняемым* файлом. Компоновщик перемещает данные и команды в объектных файлах так, чтобы они не наслаивались друг на друга. Он использует информацию из таблицы символов для коррекции адресов перемещаемых глобальных переменных и меток. Вызовите GCC для компоновки объектного файла при помощи команды

```
gcc prog.o –o prog
```

Мы снова можем дизассемблировать исполняемый файл:

```
objdump -S -t prog
```

Обратите внимание, что теперь, когда глобальным переменным `f`, `g` и `y` выделены адреса памяти, они перечислены как глобальные символы (на что указывает флаг `g`) и расположены в сегменте `.bss`, где размещаются неинициализированные глобальные переменные.

Код запуска слишком длинный, чтобы его можно было здесь показать, но обновленная таблица символов и программный код, дизассемблированный теперь из исполняемого файла, показаны ниже. Мы снова добавили подписи к интересующим нас столбцам. Теперь функции и глобальные переменные располагаются по фактическим адресам. Согласно таблице символов общий код и сегменты данных (которые включают код запуска и системные данные) начинаются с 0x10074 и 0x115e0 соответственно. Блок `func` начинается с адреса 0x10144 и имеет размер 0x3c байт (15 инструкций). Блок `main` начинается с 0x10180 и имеет размер 0x34 байт (13 инструкций). Каждая глобальная переменная имеет размер 4 байта; переменная `f` расположена по адресу памяти 0x11a30, переменная `g` – по адресу 0x11a34, а `y` – по адресу 0x11a38.

```
Адрес              Размер  Имя символа
00010074 l d .text  00000000  .text
000115e0 l d .data  00000000  .data
00010144 g F .text  0000003c  func
00010180 g F .text  00000034  main
```

```
00011a30 g O .bss   00000004  f
00011a34 g O .bss   00000004  g
00011a38 g O .bss   00000004  y
```

Обратите внимание, что размер блока `func`, приведенного ниже, теперь составляет 15 инструкций вместо 16. Вызов `func` является ближним, поэтому для него достаточно только одной инструкции `jalr`. Аналогично код `main` уменьшился с 17 до 13 инструкций из-за ближних вызовов и хранения рядом с глобальным указателем `gp`. Программа сохраняет значение в `f` с помощью одной инструкции `sw a4, -944(gp)`. Из этой инструкции мы также можем определить значение глобального указателя `gp`, которое было инициализировано кодом запуска. Мы знаем, что `f` находится по адресу 0x11a30, следовательно, начальное значение `gp` равно 0x11a30 + 944 = 0x11DE0.

```
00010144 <func>:
int f, g, y;

int func(int a, int b) {
   10144: ff010113           addi sp,sp,-16
   10148: 00112623           sw ra,12(sp)
   1014c: 00812423           sw s0,8(sp)
   10150: 00050413           mv s0,a0
  if (b<0) return (a+b);
   10154: 00a58533           add a0,a1,a0
   10158: 0005da63           bgez a1,1016c <func+0x28>
  else return(a + func(a, b-1));
}
   1015c: 00c12083           lw ra,12(sp)
   10160: 00812403           lw s0,8(sp)
   10164: 01010113           addi sp,sp,16
   10168: 00008067           ret
  else return(a + func(a, b-1));
   1016c: fff58593           addi a1,a1,-1
   10170: 00040513           mv a0,s0
   10174: fd1ff0ef           jal ra,10144 <func>
   10178: 00850533           add a0,a0,s0
   1017c: fe1ff06f           j 1015c <func+0x18>

00010180 <main>:
void main() {
   10180: ff010113           addi sp,sp,-16
   10184: 00112623           sw ra,12(sp)
  f=2;
   10188: 00200713           li a4,2
   1018c: c4e1a823           sw a4,-944(gp) # 11a30 <f>
  g=3;
   10190: 00300713           li a4,3
   10194: c4e1aa23           sw a4,-940(gp) # 11a34 <g>
  y=func(f,g);
   10198: 00300593           li a1,3
   1019c: 00200513           li a0,2
   101a0: fa5ff0ef           jal ra,10144 <func>
   101a4: c4a1ac23           sw a0,-936(gp) # 11a38 <y>
```

```
  return;
}
   101a8: 00c12083          lw ra,12(sp)
   101ac: 01010113          addi sp,sp,16
   101b0: 00008067          ret
```

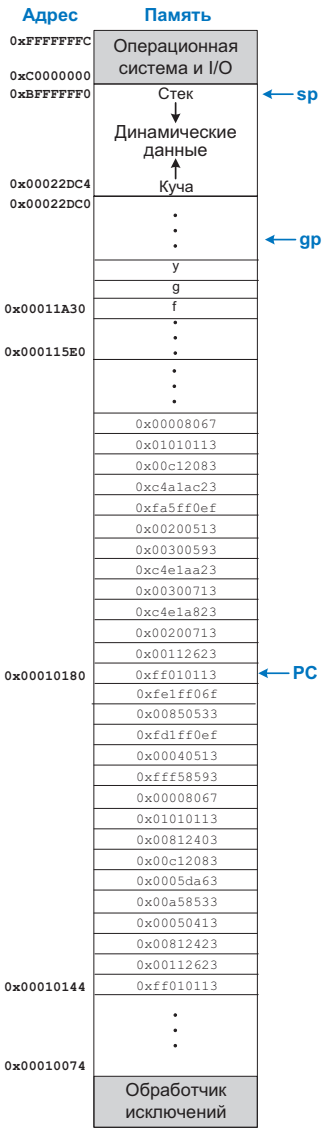


Рис. 6.34 **Исполняемый файл, загруженный в память**

6.5.6. Загрузка

Операционная система загружает программу, считывая сегмент кода исполняемого файла с устройства хранения данных (обычно это жесткий диск или флеш-память) в сегмент кода памяти. Операционная система переходит к началу программы и начинает ее выполнение. На **рис. 6.34** показана карта памяти в начале выполнения программы.

6.6. Добавочные сведения

В этом разделе рассматриваются несколько дополнительных тем, для которых не нашлось места в других частях главы. Эти темы включают порядок байтов, исключения, знаковые и беззнаковые арифметические инструкции, инструкции с плавающей запятой и сжатые (16-битные) инструкции.

6.6.1. Порядок байтов

Память с побайтовой адресацией может быть организована с *прямым порядком* следования байтов (от младшего к старшему; little-endian) или с *обратным порядком* (от старшего к младшему; big-endian), как показано на **рис. 6.35**. В обоих случаях *самый старший байт* (most significant byte, MSB) находится слева, а *самый млад-*

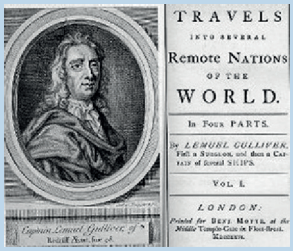
Рис. 6.35 **Адресация памяти с прямым и обратным порядками байтов**

ший байт (least significant byte, LSB) – справа. Пословная адресация одинакова в обеих моделях, то есть один и тот же адрес слова указывает на одни и те же четыре байта. Различаются только *адреса байтов* внутри слова (**рис. 6.35**). В системах с прямым порядком следования байты пронумерованы от 0, начиная с самого младшего байта. В системах с обратным порядком следования байты пронумерованы от 0, начиная с самого старшего байта.

Архитектура RISC-V обычно использует прямой порядок байтов, хотя существует вариант с обратным порядком байтов. Архитектура IBM PowerPC (ранее применявшаяся в компьютерах Macintosh) использует адресацию с обратным порядком байтов. Архитектура Intel x86 (которая применяется в привычных нам ПК) использует адресацию с прямым порядком байтов. Выбор порядка байтов является совершенно произвольным, но приводит к проблемам при совместном использовании одних и тех же данных между компьютерами, принадлежащими к архитектурам с разным порядком байтов. В примерах этой книги мы используем формат с прямым порядком байтов, когда этот порядок имеет значение.

6.6.2. Исключения

Исключение (exception) подобно незапланированному вызову функции, вызванному аппаратным или программным событием. Например, процессор может получить уведомление о том, что пользователь нажал клавишу на клавиатуре. В этом случае процессор может приостановить выполнение программы, определить нажатую клавишу и сохранить информацию об этом, после чего возобновить выполнение прерванной программы. Исключения, вызванные устройствами ввода-вывода, такими как клавиатура, часто называют *прерываниями* (interrupt). С другой стороны, исключение может быть вызвано ошибкой в программе, например, из-за использования неопределенной команды. В этом случае программа совершает переход к коду операционной системы (ОС), который может завершить выполнение программы-нарушителя. Исключения, возникающие в программах, иногда называют *ловушками* (traps). Другими причинами исключений могут быть деление на ноль, попытка чтения несуществующей памяти, аппаратные сбои, точка останова отладчика (debugger breakpoint) и арифметическое


TRAVELS
INTO SEVERAL
Remote NATIONS
OF THE
WORLD.
In FOUR PARTS.
VOL. I.
LONDON:

Происхождение английских терминов *little-endian* (прямой порядок следования байтов) и *big-endian* (обратный порядок) восходит к произведению «Путешествия Гулливера» Джонатана Свифта, впервые опубликованному в 1726 году под псевдонимом Исаака Бикерстаффа. В его рассказах король лилипутов требовал от граждан (остроконечников, англ. Little-Endians) разбивать яйцо с острого конца. Тупоконечники (англ. Big-Endians) были повстанцами, разбивавшими яйца с тупого конца. Термины были впервые применены к компьютерным архитектурам Дэнни Коэном в его статье «О священных войнах и мольбе о мире», опубликованной в День дурака 1 апреля 1980 года (USC/ISI IEN 137). (Фотография из коллекции Бразертона любезно предоставлена библиотекой Лидского университета.)

Существует четвертый уровень привилегий, называемый *режимом гипервизора* (hypervisor mode, H-режим), который поддерживает *виртуализацию машин*, то есть существование нескольких виртуальных машин (потенциально с несколькими операционными системами), работающих на одной физической машине. H-режим имеет более высокие привилегии, чем S-режим, но не такие, как M-режим.

Архитектура RISC-V определяет множество CSR, и все они должны быть инициализированы при запуске.

переполнение. Как и любая другая вызываемая функция, исключение должно сохранить адрес возврата, перейти на какой-либо адрес, выполнить свою работу, очистить флаг исключения и вернуться в программу на то место, где она прервала свою работу.

Режимы выполнения и уровни привилегий

Процессор RISC-V может работать в одном из нескольких *режимов выполнения* (execution mode) с разными *уровнями привилегий*. Уровни привилегий определяют, какие инструкции может выполнить процессор и к какой памяти он может получить доступ. Три основных уровня привилегий в архитектуре RISC-V в порядке увеличения – это пользовательский режим, режим супервизора и машинный режим. *Машинный режим* (M-режим) – это наивысший уровень привилегий; программа, работающая в этом режиме, может получить доступ ко всем регистрам и ячейкам памяти. M-режим – это единственный режим привилегий, используемый в процессорах, работающих без операционной системы (ОС), включая многие встраиваемые системы. Пользовательские приложения, которые работают поверх ОС, обычно работают в *пользовательском режиме* (U-режим), а ОС работает в режиме *супервизора* (S-режим). Пользовательские программы не имеют доступа к привилегированным регистрам или ячейкам памяти, зарезервированным для ОС. В этом и заключается смысл использования разных режимов – они защищают состояние системы от повреждения. В данном учебнике мы рассматриваем исключения, возникающие при работе в M-режиме. Исключения, возникающие на других уровнях, аналогичны, но используют регистры, связанные с соответствующим режимом.

Значение `mcause` можно классифицировать как прерывание или исключение в соответствии с крайним левым столбцом в табл. 6.6, который содержит бит 31 из `mcause`. Биты [30:0] из `mcause` содержат *код исключения*, указывающий причину прерывания или исключения.

Исключения могут использовать один из двух режимов обработки исключений: *прямой* или *векторный*. В архитектуре RISC-V обычно применяется прямой режим, когда все исключения переходят к одному и тому же базовому адресу, закодированному в битах 31:2 `mtvec`. В векторном режиме исключения переходят по смещению от базового адреса в зависимости от причины исключения. Адреса векторных обработчиков исключений разделяются небольшими интервалами, например 32 байта, поэтому для обработки исключения программе иногда приходится совершать еще один переход к более крупному обработчику. Режим исключения закодирован в битах 1:0 `mtvec`; 002 означает прямой режим, 012 – векторный.

Обработчики исключений

Обработчики исключений при выполнении своей работы используют четыре специальных регистра, называемых *регистрами управления и состояния* (control and state register, CSR): `mtvec`, `mcause`, `mepc` и `mscratch`. Регистр базового адреса вектора ловушек, `mtvec`, содержит адрес обработчика исключений. Когда возникает исключение, процессор записывает причину исключения в `mcause` (**табл. 6.6**), сохраняет в `mepc` значение счетчика команд для инструкции, которая вызвала исключение, и переходит к обработчику исключения по адресу, предварительно указанному в `mtvec`.

Перейдя по адресу в mtvec, обработчик исключений читает регистр mcause, чтобы выяснить, что вызвало исключение, и реагирует соответствующим образом (например, считывая код нажатой клавиши при аппаратном прерывании).

Затем он либо прерывает выполнение программы, либо возвращается в программу, выполняя mret, инструкцию возврата из машинного исключения, которая переходит к адресу, сохраненному в mepc. Сохранение в mepc адреса инструкции, которая вызвала исключение, аналогично использованию регистра ra для сохранения адреса возврата во время выполнения инструкции jal. Обработчики исключений используют программные регистры (x1-x31), поэтому они применяют область памяти, на которую указывает mscratch, для сохранения и восстановления этих регистров.

Перечень регистров, связанных с исключениями, зависит от режима работы. Регистры M-режима — это mtvec, mepc, mcause и mscratch, а регистры S-режима — sepc, scause и sscratch. Для H-режима также есть свои регистры. Отдельные регистры исключений, выделенные для каждого режима, обеспечивают аппаратную поддержку нескольких уровней привилегий.

Таблица 6.6 Коды наиболее частых причин исключения

Прерывание	Код исключения	Описание
1	3	Машинное программное прерывание
1	7	Машинное прерывание по таймеру
1	11	Машинное внешнее прерывание
0	0	Неверный адрес инструкции
0	2	Недопустимая инструкция
0	3	Точка останова
0	4	Неверный адрес загрузки
0	5	Сбой загрузки
0	6	Неверный адрес сохранения
0	7	Сбой сохранения
0	8	Внешний вызов в U-режиме
0	9	Внешний вызов в S-режиме
0	11	Внешний вызов в M-режиме

Инструкции, связанные с исключениями

Обработчики исключений используют специальные инструкции для обработки исключений. Эти инструкции называются *привилегированными*, поскольку они обращаются к CSR. Они являются частью базового набора инструкций RV32I (**приложение B**, **табл. B.8**). Регистры mepc и mcause не являются частью программных регистров RISC-V (x1–x31), поэтому обработчик исключений должен переместить эти регистры специального назначения (special purpose register, CSR) в программные

`csrrw` — это обычная инструкция RISC-V (табл. B.8 в приложении B), но `csrr` и `csrw` — это псевдоинструкции. Псевдоинструкция `csrr` реализована как `csrrs rd, CSR, x0` а `csrw` реализована как `csrrw x0, CSR, rs1`.

регистры для чтения и работы с ними. Набор инструкций RISC-V содержит три инструкции для чтения, записи или чтения и записи CSR: `csrr` (чтение CSR), `csrw` (запись CSR) и `csrrw` (чтение/запись CSR). Например, инструкция `csrr t1, mcause` считывает значение из `mcause` в `t1`; инструкция `csrw mepc, t2` записывает значение `t2` в `mepc`; иинструкция `csrrw t1, mscratch, t0` одновременно считывает значение из `mscratch` в `t1` и записывает значение из `t0` в `mscratch`.

Промежуточный итог

Когда процессор обнаруживает исключение, он:

1) переходит к адресу обработчика исключений, хранящемуся в `mtvec`;
2) обработчик исключений сохраняет регистры в небольшом стеке, на который указывает `mscratch`, а затем использует псевдоинструкцию `csrr` (чтение CSR) для выяснения причины исключения (записанной в виде кода в `mcause`) и соответствующего ответа;
3) когда обработчик завершает работу, он необязательно увеличивает `mepc` на 4, восстанавливает регистры из памяти и либо прерывает программу, либо возвращается к пользовательскому коду с помощью инструкции `mret`, которая переходит на адрес, сохраненный в `mepc`.

При запуске процессор переходит к *Вектору исключения сброса*, жестко заданному адресу аппаратной памяти, например 0x200, который является начальным адресом кода *загрузчика* boot loader), также называемого *загрузочным кодом* (boot code). Хотя сброс не является типичным исключением, возникающим во время выполнения программы, его относят к таковым, потому что сброс — это исключительное состояние процессора. Загрузочный код настраивает систему памяти, инициализирует CSR и указатель стека и считывает часть ОС с жесткого диска. Затем начинается гораздо более длительный процесс загрузки ОС. В конечном итоге ОС загрузит программу, перейдет в непривилегированный пользовательский режим и запустит программу. В системах с «голым железом», то есть не имеющих операционной системы, пользовательский код (возможно, с облегченным загрузочным кодом для установки указателя стека и т. д.) обычно помещается непосредственно по адресу вектора сброса.

Пример 6.7 ОБРАБОТЧИК ИСКЛЮЧЕНИЙ

Разработайте обработчик исключений для работы со следующими двумя исключениями: недопустимая инструкция (`mcause` = 2) и неверный адрес загрузки (`mcause` = 4). Если возникает недопустимая инструкция, программа должна просто продолжить выполнение после недопустимой инструкции. При возникновении исключения, связанного с недопустимым адресом загрузки, программа должна остановиться. Если возникает какое-либо другое исключение, программа должна попытаться повторно выполнить инструкцию.

Решение Обработчик исключений начинает с сохранения регистров программы, которые будут перезаписаны. Затем он проверяет причину исключения и (1) продолжает выполнение сразу после исключения по недопустимой инструкции (т. е. переходит по адресу `mepc + 4`), (2) прерывает выполнение программы при исключении по недопустимому адресу загрузки или

(3) пытается повторно выполнить команду, которая вызвала исключение (т. е. возвратиться по адресу в mepc) при любом другом исключении. Перед тем как вернуться в программу, обработчик восстанавливает все перезаписанные регистры. Чтобы прервать выполнение программы, обработчик переходит к коду выхода, расположенному по адресу метки выхода (в примере не показан). В программах, работающих поверх ОС, команду выхода j можно заменить вызовом среды (ecall) с кодом возврата, хранящимся в программном регистре, например a0.

Особенно важным исключением является *системный вызов*, также называемый *обращением к ОС*. Программы используют их для вызова функции в ОС, которая работает с более высоким уровнем привилегий, чем код пользователя. Это исключение инициируется пользовательской программой, выполняющей инструкцию ecall. Как и при вызове функции, программа может настраивать регистры аргументов перед выполнением системного вызова.

```
# сохранение регистров, которые будут перезаписаны
  csrrw t0, mscratch, t0    # поменять местами t0
                            # и mscratch
  sw t1, 0(t0)              # сохранить t1 в стек mscratch
  sw t2, 4(t0)              # сохранить t2 в стек mscratch

# проверка причины исключения
  csrr t1, mcause           # t1 = mcause
  addi t2, x0, 2            # t2 = 2 (недопустимая инструкция)

illegalinstr:
  bne t1, t2, checkother    # переход, если инструкция допустимая
  csrr t2, mepc             # t2 = счетчик команд исключения
  addi t2, t2, 4            # увеличить счетчик команд на 4
  csrw mepc, t2             # mepc = mepc + 4
  j done                    # восстановление регистров и возврат

checkother:
  addi t2, x0, 4            # t2 = 4 (недопустимый адрес загрузки)
  bne t1, t2, done          # переход, если адрес допустимый
  j exit                    # выход из программы
# восстановление регистров и возврат из исключения
done:
  lw t1, 0(t0)              # восстановить t1 из стека mscratch
  lw t2, 4(t0)              # восстановить t1 из стека mscratch
  csrrw t0, mscratch, t0    # поменять местами t0 и mscratch
  mret                      # возврат в программу (PC = mepc)
  ...
exit:
  ...
```

6.6.3. Команды для чисел со знаком и без знака

Напомним, что двоичное число может быть со знаком или без знака. Как и большинство архитектур, для представления чисел со знаком RISC-V использует дополнительный код. Некоторые команды RISC-V имеют две версии – одну для чисел со знаком и вторую для чисел без знака. Примером таких команд являются команды сложения и вычитания, умножения и деления, команды сравнения и команды загрузки части слова.

В отличие от других архитектур, таких как MIPS и ARM, система инструкций RISC-V не содержит инструкции (или исключения) для обнаружения переполнения, поскольку его можно обнаружить с помощью других инструкций. Например, следующий код обнаруживает переполнение без знака при сложении t1 и t2:

```
add  t0, t1, t2
bltu t0, t1, overflow
```

Другими словами, если результат (t0) меньше любого из операндов (в данном случае t1), значит, произошло переполнение.

Следующий код обнаруживает переполнение при сложении двух чисел со знаком, t1 и t2:

```
add  t0, t1, t2
slti t3, t2, 0
slt  t4, t0, t1
bne  t3, t4, overflow
```

В форме уравнения переполнение можно записать так:

```
overflow = (t2 < 0) & (t0 ≥ t1) | (t2 ≥ 0) & (t0 < t1)
```

Это означает, что переполнение происходит, когда один операнд отрицательный (t3 = 1) и результат не меньше, чем другой операнд (t4 = 0), или когда один операнд больше или равен 0 (t3 = 0) и результат меньше, чем другой операнд (t4 = 1).

Умножение и деление

Результаты операций умножения и деления зависят от того, учитывают они знак или нет. Например, если интерпретировать 0xFFFFFFFF как число без знака, то оно будет представлять собой очень большую величину, но как знаковое число оно имеет значение −1. Следовательно, произведение 0xFFFFFFFF × 0xFFFFFFFF равно 0xFFFFFFFE00000001, если используются беззнаковые числа, и 0x0000000000000001 при использовании чисел со знаком. (Обратите внимание, что младшие 32 бита одинаковы как для знакового, так и для беззнакового умножения.) Поэтому инструкции умножения и деления бывают в двух версиях – для знаковых и беззнаковых чисел. Инструкции `mulh` и `div` обрабатывают операнды как числа со знаком. Инструкции `mulhu` и `divu` обрабатывают операнды как беззнаковые числа. Инструкция `mulhsu` рассматривает первый операнд как знаковый, а второй как беззнаковый. Все команды умножения старших разрядов (`mulh`, `mulhu` и `mulhsu`) помещают 32 старших разряда в регистр-назначение `rd`. Младшие 32 бита результата не отличаются для беззнакового и знакового умножений, поэтому инструкция `mul` помещает младшие 32 бита результата умножения в регистр `rd` как в случае беззнакового, так и знакового умножения.

Инструкция set less than

Инструкция *определения меньшего среди двух аргументов* (set less than) предназначена для сравнения либо двух регистров (`slt`), либо регистра и константы (`slti`). Эта инструкция тоже существует в знаковой (`slt` и `slti`) и беззнаковой (`sltu` и `sltiu`) версиях. В сравнении со знаком 0x80000000 меньше любого другого числа, потому что это максимальное отрицательное число в дополнительном коде. В беззнаковом сравнении 0x80000000 больше 0x7FFFFFFF, но меньше 0x80000001, потому что все числа положительны. Имейте в виду, что инструкция `sltiu` дополняет 12-битное значение знаковым битом непосредственно перед тем, как рассматривать его как беззнаковое число. Например, инструкция `sltiu s0, s1, -1273` сравнивает `s1` с 0xFFFFFB07, рассматривая константу как большое положительное число.

Условный переход

Инструкции перехода по условиям «если меньше» и «если больше» также существуют в знаковой (`blt` и `bge`) и беззнаковой (`bltu` и `bgeu`)

версиях. Знаковые версии рассматривают два исходных операнда как два числа в дополнительном коде, а беззнаковые версии рассматривают исходные операнды как числа без знака.

Загрузка

Как описано в **разделе 6.3.6**, инструкции загрузки байта бывают в версиях со знаком (`lb`) и без знака (`lbu`). Инструкция `lb` дополняет байт знаковым битом, а `lbu` дополняет байт нулями, заполняя весь 32-битный регистр. Аналогичным образом инструкции загрузки полуслова со знаком и без знака (`lh` и `lhu`) загружают два байта в нижнюю половину слова и дополняют их до полного слова битом знака или нулем соответственно.

6.6.4. Команды для работы с числами с плавающей запятой

В архитектуре RISC-V предусмотрены дополнительные расширения для работы с числами с плавающей запятой под названиями RVF, RVD и RVQ, предназначенные для работы с числами одинарной, двойной и четверной точности соответственно. Расширения RVF/D/Q определяют 32 регистра с плавающей запятой, от `f0` до `f31`, с шириной 32, 64 или 128 бит соответственно. Когда процессор реализует несколько расширений с плавающей запятой, он использует нижнюю часть регистра с плавающей запятой для инструкций с более низкой точностью. Регистры `f0-f31` отделены от программных (также называемых целочисленными) регистров `x0-x31`. Как и в случае с программными регистрами, регистры с плавающей запятой по соглашению зарезервированы для определенных целей, как показано в **табл. 6.7**.

Таблица 6.7 Набор регистров с плавающей запятой в архитектуре RISC-V

Обозначение	Номер регистра	Применение
ft0–7	f0–7	Временные переменные
fs0–1	f8–9	Сохраненные переменные
fa0–1	f10–11	Аргументы функции / Возвращаемые значения
fa2–7	f12–17	Аргументы функции
fs2–11	f18–27	Сохраненные переменные
ft8–11	f28–31	Временные переменные

В **табл. B.3** в **приложении B** перечислены все инструкции с плавающей запятой. Инструкции вычисления и сравнения используют одну и ту же мнемонику для всех значений точности с добавлением суффикса `.s`, `.d` или `.q` в конце для обозначения точности. Например, инструкции

`fadd.s`, `fadd.d` и `fadd.q` выполняют сложение с одинарной, двойной и четверной точностью соответственно. Для вычислений с плавающей запятой предназначены также инструкции `fsub`, `fmul`, `fdiv`, `fsqrt`, `fmadd` (умножение с накоплением) и `fmin`. Доступ к памяти устроен несколько иначе – здесь существуют отдельные инструкции для каждой точности. Загрузку выполняют инструкции `flw`, `fld` и `flq`, а сохранение – `fsw`, `fsd` и `fsq`.

Инструкции с плавающей запятой используют форматы типа *R*, *I* и *S*, а также новый формат типа *R4* (**рис. B.1** в **приложении B**). Этот формат необходим для инструкций умножения с накоплением, которые используют четыре регистровых операнда. **Пример кода 6.31** представляет собой измененный **пример кода 6.21** для работы с массивом оценок (score) с плавающей запятой одинарной точности. Изменения выделены жирным шрифтом.

Пример кода 6.31 ИСПОЛЬЗОВАНИЕ ЦИКЛА FOR ДЛЯ ДОСТУПА К МАССИВУ ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Код на языке высокого уровня

```
int i;
float scores[200];

for (i = 0; i < 200; i = i + 1)


scores[i] = scores[i] + 10;
```

Код на языке ассемблера RISC-V

```
# s0 = базовый адрес оценок, s1 = i

  addi     s1, zero, 0   # i = 0
  addi     t2, zero, 200 # t2 = 200
  addi     t3, zero, 10  # t3 = 10
  fcvt.s.w ft0, t3       # ft0 = 10.0

for:
  bge      s1, t2, done  # если i >= 200, завершить
  slli     t3, s1, 2     # t3 = i * 4
  add      t3, t3, s0    # адрес scores[i]
  flw      ft1, 0(t3)    # ft1 = scores[i]
  fadd.s   ft1, ft1, ft0 # ft1 = scores[i] + 10
  fsw      ft1, 0(t3)    # scores[i] = t1
  addi     s1, s1, 1     # i = i + 1
  j        for           # повтор
done:
```

6.6.5. Сжатые инструкции

Расширение *сжатых инструкций* (compressed instruction extension, RVC) RISC-V уменьшает размер обычных целочисленных инструкций и инструкций с плавающей запятой до 16 бит за счет уменьшения размеров полей управления, констант и регистров, а также за счет использования *избыточных* или *подразумеваемых* регистров. Уменьшение размера команд снижает аппаратные затраты, потребляемую электрическую мощность и требуемый объем памяти – все это чрезвычайно важно для

карманных и мобильных приложений. Согласно Руководству по набору инструкций RISC-V, обычно от 50 % до 60 % инструкций программы могут быть заменены инструкциями RVC. 16-битные инструкции по-прежнему работают с базовым размером данных (32, 64 или 128 бит), как определено базовым набором команд. Программы на ассемблере могут использовать как сжатые, так и 32-битные инструкции, если процессор может обрабатывать и те, и другие.

Многие ассемблеры RISC-V генерируют код на основе смеси сжатых и несжатых инструкций, используя сжатые инструкции везде, где это возможно, чтобы минимизировать размер кода.

Большинство инструкций RV32I имеют сжатый аналог, который начинается с префикса `c.`, как показано в **табл. B.6 приложения B**. Чтобы уменьшить размер, в большинстве сжатых инструкций указывают только два регистра: первый регистр-источник также является регистром-назначением. В большинстве инструкций применяются 3-битные коды регистров для указания одного из 8 регистров `x8-x15`. Регистр `x8` кодируется как 0002, `x9` как 0012 и т. д. Константы тоже короче (6–11 бит), и для кодов операций доступно меньшее количество битов. На **рис. B.2** в **приложении B** показаны сжатые форматы команд.

Пример кода 6.32 представляет собой измененный **пример кода 6.21**, в котором применяются сжатые инструкции. Обратите внимание, что константа 200 слишком велика, чтобы поместиться в сжатую инструкцию, поэтому регистр `s0` инициализируется с помощью несжатой инструкции `addi`. Сжатой инструкции `c.bge` не существует, поэтому также используется несжатая версия `bge`. Мы также инкрементируем `s0` в качестве указателя на элементы `scores[i]`, потому что сжатые инструкции с двумя операндами имеют ограниченные возможности сдвига и сложения. В итоге нам удалось уменьшить программу с 40 до 22 байт.

Пример кода 6.32 ИСПОЛЬЗОВАНИЕ СЖАТЫХ ИНСТРУКЦИЙ

Код на языке высокого уровня

```
int i;
int scores[200];

for (i = 0; i < 200; i = i + 1)

    scores[i] = scores[i] + 10;
```

Код на языке ассемблера RISC-V

```
# s0 = базовый адрес массива scores, s1 = i

  c.li s1, 0           # i = 0
  addi t2, zero, 200   # t2 = 200

for:
  bge s1, t2, done     # если i >= 200, то завершить
  c.lw a3, 0(s0)       # a3 = scores[i]
  c.addi a3, 10        # a3 = scores[i] + 10
  c.sw a3, 0(s0)       # scores[i] = a3
  c.addi s0, 4         # следующий элемент scores
  c.addi s1, 1         # i = i + 1
  c.j for              # повтор
done:
```

Архитектура RISC-V описана в Руководстве по набору команд RISC-V (http://riscv.org/specifications). Ранние версии руководства, вплоть до версии 2.2, представляют собой пример отличной документации — краткой, удобочитаемой и снабженной логическим обоснованием проектных решений, воплощенных в архитектуре.

6.7. Эволюция архитектуры RISC-V

Архитектура RISC-V была разработана как коммерчески востребованная компьютерная архитектура с открытым исходным кодом, которая является надежной, эффективной и гибкой. RISC-V отличается от других архитектур, поскольку имеет открытый исходный код, использует базовые наборы инструкций для облегчения совместимости, поддерживает полный спектр микроархитектур, от встраиваемых систем до высокопроизводительных компьютеров, предлагает как статичные, так и настраиваемые расширения, а также предоставляет такие преимущества, как сжатые инструкции и набор инструкций RV128I, которые позволяют оптимизировать аппаратную основу и поддерживают как существующие, так и будущие разработки, обеспечивая долговечность архитектуры.

Вокруг RISC-V сформировалось сообщество промышленных и научных партнеров RISC-V International (http://riscv.org), тем самым ускорив инновации и коммерциализацию. Этот консорциум разработчиков также помогает проектировать и ратифицировать архитектуру RISC-V. Сообщество RISC-V International к 2021 году насчитывает более 500 членов как из академических, так и из промышленных кругов, включая Western Digital, NVIDIA, Microchip и Samsung.

6.7.1. Базовые наборы команд и расширения RISC-V

Архитектура RISC-V содержит различные базовые наборы команд и расширения, поэтому она может поддерживать широкий спектр оборудования — от небольших недорогих встраиваемых процессоров, например в портативных устройствах, до высокопроизводительных, многоядерных, многопоточных систем. RISC-V имеет 32-, 64- и 128-битные базовые наборы инструкций: RV32I/E, RV64I и RV128I соответственно. 32-битный базовый набор команд входит в стандартную версию RV32I, которую мы рассматриваем в этой главе, и во встраиваемую версию RV32E всего с 16 регистрами, предназначенную для очень недорогих процессоров. С 2021 г. зафиксированы только наборы команд RV32I и RV64I; наборы RV32E и RV128I все еще находятся в стадии разработки. Наряду с этими базовыми архитектурами спецификация RISC-V также определяет расширения, перечисленные в табл. 6.8. Наиболее часто используемые расширения — операции с плавающей запятой (RVF/D/Q), сжатые инструкции (RVC) и атомарные инструкции (RVA) — полностью

определены и зафиксированы, чтобы обеспечить возможность разработки и коммерциализации оборудования. Остальные расширения все еще находятся в разработке.

Таблица 6.8 Расширения RISC-V

Расширение	Описание	Статус
M	Целочисленное умножение и деление	Зафиксировано
F	Вычисления с плавающей запятой одинарной точности	Зафиксировано
D	Вычисления с плавающей запятой двойной точности	Зафиксировано
Q	Вычисления с плавающей запятой четверной точности	Зафиксировано
C	Сжатые инструкции	Зафиксировано
A	Атомарные инструкции	Зафиксировано
B	Побитовые операции	Разработка
L	Десятичные операции с плавающей запятой	Разработка
J	Динамически транслируемые языки	Разработка
T	Транзакционная память	Разработка
P	Упакованные инструкции SIMD	Разработка
V	Векторные операции	Разработка

Все процессоры RISC-V должны поддерживать одну из базовых архитектур – RV32/64/128I или RV32E – и могут дополнительно поддерживать расширения, такие как сжатые инструкции или операции с плавающей запятой. За счет использования расширений вместо новых версий архитектуры RISC-V снижается сложность организации обратной или прямой совместимости между микроархитектурами. Все процессоры должны поддерживать как минимум базовую архитектуру. Но процессор не обязан поддерживать все (или даже какие-либо) расширения.

Чтобы понять эволюцию архитектуры RISC-V, важно понимать другие архитектуры, предшествовавшие RISC-V, и особенно архитектуру MIPS. RISC-V следует многим принципам архитектуры MIPS, но при этом выигрывает с точки зрения современных архитектур и приложений, включая такие специфические применения, как встроенные, многоядерные и многопоточные системы, и обладает хорошей расширяемостью. В следующем разделе мы сравним архитектуры RISC-V и MIPS.

6.7.2. Сравнение архитектур RISC-V и MIPS

Архитектура RISC-V имеет много общего с архитектурой MIPS, разработанной Джоном Хеннесси в 1980-х годах, но она устраняет некото-

рые ненужные сложности – и тут же вводит новые, например инструкции с разбросанными по всей команде битами константы. К сходству архитектур можно отнести форматы ассемблера и машинного кода, мнемонику инструкций, именование регистров, а также соглашения о стеках и вызовах. Различия заключаются в размерах констант и кода команд RISC-V, условных переходов относительно значения в PC (вместо PC + 4), когда условные и безусловные переходы выполняются относительно PC, в отсутствии слота задержки перехода, который есть в MIPS, в строгом определении полей инструкций регистра источника и назначения, в различном количестве временных, оберегаемых регистров и регистров аргументов, а также в большей расширяемости за счет включения большего количества управляющих битов в инструкцию. Сохраняя регистровые операнды `rs1`, `rs2` и `rd` в одних и тех же битовых полях каждого типа инструкций, который их использует, RISC-V упрощает аппаратную структуру декодера по сравнению с MIPS. Точно так же упрощает аппаратное обеспечение и своеобразное кодирование констант RISC-V.

6.7.3. Сравнение архитектур RISC-V и ARM

ARM – это архитектура RISC, которая была разработана в 1980-х годах примерно в то же время, что и архитектура MIPS. За последнее десятилетие процессоры ARM заняли доминирующее положение на рынке мобильных устройств, а также используются в других приложениях, таких как роботы, игровые автоматы и серверы. Сходство ARM с RISC-V заключается в небольшом количестве форматов машинного кода и команд ассемблера, а также схожих соглашениях о стеке и вызовов функций. Архитектура ARM отличается от RISC-V поддержкой условного выполнения, сложными режимами индексирования для доступа к памяти, способностью вставлять и извлекать несколько регистров в стек с помощью одной инструкции, необязательным смещением регистров-источников и нетрадиционным кодированием констант. Значения констант кодируются сочетанием 8-битного числового значения и 4-битного показателя вращения, и они кодируют только положительные константы (вычитание определяется управляющими битами). Некоторые особенности ARM – в частности, условное выполнение, смещенные регистры и режимы индексации – обычно присущи только архитектурам CISC, но ARM поддерживает их для уменьшения размера программы и, следовательно, размера памяти, что критично для встраиваемых и портативных устройств. При этом эти конструктивные решения также приводят к усложнию схемы процессора.

6.8. Живой пример: архитектура x86

Практически все персональные компьютеры используют процессоры с архитектурой x86. Архитектура x86, также называемая IA-32, – это 32-разрядная архитектура, изначально разработанная компанией Intel. Компания AMD продает и x86-совместимые микропроцессоры.

Архитектура x86 имеет долгую и запутанную историю, которая берет начало в 1978 году, когда Intel объявила о разработке 16-битного микропроцессора 8086. Компания IBM выбрала 8086 и его брата 8088 для своих первых персональных компьютеров (ПК). В 1985 году Intel представила 32-разрядный микропроцессор 80386, который был обратно совместим с 8086 и мог запускать программы, разработанные для более ранних ПК. Процессорные архитектуры, совместимые с 80386, называют x86-совместимыми архитектурами. Процессоры Pentium, Core и Athlon – наиболее известные x86-совместимые процессоры.

Различные группы разработчиков в Intel и AMD на протяжении многих лет добавляли множество новых команд и возможностей в устаревшую архитектуру. В результате она выглядит гораздо менее элегантно, чем RISC-V. Как объясняют Паттерсон и Хеннесси: «эта архитектура похожа на лоскутное одеяло, ее сложно понять и невозможно полюбить». Тем не менее совместимость программного обеспечения гораздо важнее технической элегантности, так что x86 является де-факто стандартом для ПК на протяжении более чем двух десятилетий. Каждый год продается свыше 100 млн x86-совместимых микропроцессоров. Это огромный рынок, оправдывающий ежегодные затраты на улучшение этих процессоров, превышающие 5 млрд долларов.

Архитектура x86 является примером CISC-архитектуры (Complex Instruction Set Computer – компьютер с полным набором команд). В отличие от команд в RISC-архитектурах, таких как RISC-V, каждая CISC-команда способна произвести больше работы. Из-за этого программы для CISC-архитектур обычно состоят из меньшего количества команд. Коды команд были подобраны так, чтобы обеспечивать наибольшую компактность кода – это требовалось в те времена, когда стоимость оперативной памяти была гораздо выше, чем сейчас. Команды имеют переменную длину, которая зачастую меньше 32 бит. Недостаток такого подхода состоит в том, что сложные команды трудно дешифровать, к тому же они, как правило, работают медленнее.

В этом разделе мы ознакомимся с архитектурой x86. Наша цель состоит не в том, чтобы сделать вас программистом на языке ассемблера x86, а, скорее, в том, чтобы проиллюстрировать некоторые сходства и различия между x86 и RISC-V. Мы считаем, что это интересно – посмотреть, как работает архитектура x86. Тем не менее изучение материалов из этого раздела является необязательным, чтобы понять оставшуюся часть книги. Основные различия между x86 и RISC-V (RV32I) приведены в **табл. 6.9**.

Таблица 6.9 Основные различия между RISC-V (RV32I) и x86

Характеристики	RISC-V	x86
Количество регистров	32, общего назначения	8, некоторые ограничения по использованию
Количество операндов	3 (2 источника, 1 назначение)	2 (1 источник, 1 источник/назначение)
Расположение операндов	Регистры или непосредственные операнды	Регистры, непосредственные операнды или память
Размер операнда	32 бита	8, 16 или 32 бита
Коды условий	Нет	Да
Типы команд	Простые	Простые и сложные
Размер команд	Фиксированный, 4 байта	Переменный, 1–15 байт

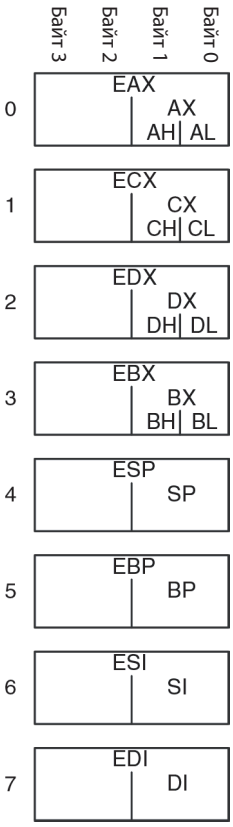


Рис. 6.36 Регистры архитектуры x86

6.8.1. Регистры x86

У микропроцессора 8086 было восемь 16-битных регистров. Некоторые из них позволяли осуществлять доступ отдельно к старшим и младшим восьми битам. Когда была представлена 32-битная архитектура 80386, регистры были просто расширены до 32 бит. Эти регистры называются `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` и `EDI`. Для обеспечения обратной совместимости была оставлена возможность получить отдельный доступ к их младшим 16 битам, а для некоторых регистров – и к двум младшим байтам, как показано на **рис. 6.36**.

Эти восемь регистров можно, за некоторым исключением, считать регистрами общего назначения. Некоторые команды не могут использовать некоторые из них. Другие команды всегда записывают результат в определенные регистры. Так же как регистр `sp` в RISC-V, регистр `ESP` обычно зарезервирован для указателя стека.

Счетчик команд в архитектуре x86 называется `EIP` (extended instruction pointer, расширенный указатель команд). Аналогично счетчику команд в архитектуре RISC-V, он увеличивается при переходе от одной команды к другой, а также может быть изменен командами условных и безусловных переходов и вызова функций.

6.8.2. Операнды x86

Команды RISC-V всегда производят действия либо с регистрами, либо с непосредственными операндами. Для перемещения данных между памятью и регистрами необходимы явные команды загрузки и сохранения. Команды x86, напротив, могут работать как с регистрами и непосредственными операндами, так и с внешней памятью. Это частично компенсирует недостаток небольшого набора регистров.

Команды RISC-V обычно определяют три операнда: два операнда-источника и один операнд-назначение. Команды x86 содержат только два операнда: операнд-источник и операнд-источник/назначение. Следовательно, команда x86 всегда записывает результат на место одного из операндов. В **табл. 6.10** перечислены поддерживаемые комбинации расположения операндов в командах x86. Из таблицы следует, что возможны любые комбинации, исключая память-память.

Таблица 6.10 Расположение операндов

Источник/Назначение	Источник	Пример	Выполняемая функция
Регистр	Регистр	`add EAX, EBX`	`EAX <- EAX + EBX`
Регистр	Непосредственный операнд	`add EAX, 42`	`EAX <- EAX + 42`
Регистр	Память	`add EAX, [20]`	`EAX <- EAX + Mem[20]`
Память	Регистр	`add [20], EAX`	`Mem[20] <- Mem[20] + EAX`
Память	Непосредственный операнд	`add [20], 42`	`Mem[20] <- Mem[20] + 42`

Аналогично RISC-V (RV32I), архитектура x86 имеет 32-битное пространство памяти с побайтовой адресацией. Но, в отличие от RISC-V, x86 поддерживает намного больше различных режимов адресации памяти. Расположение ячейки памяти задается при помощи комбинации регистра базового адреса, регистра смещения и регистра масштабируемого индекса (**табл. 6.11**). Смещение может иметь 8-, 16- или 32-битное значение. Регистр масштабируемого индекса может быть умножен на 1, 2, 4 или 8. Режим базовой адресации со смещением аналогичен режиму базовой адресации в RISC-V, используемому для команд загрузки и сохранения. Масштабируемый индекс обеспечивает простой способ доступа к массивам и структурам с 2-, 4- или 8-байтовыми элементами без необходимости использовать команды для явного расчета адресов. В то время как RISC-V оперирует с 32-битными словами данных, команды x86 могут использовать 8-, 16- или 32-битные данные. Это проиллюстрировано в **табл. 6.12**.

Таблица 6.11 Режимы адресации памяти

Пример	Назначение	Комментарий
`add EAX, [20]`	`EAX <- EAX + Mem[20]`	Смещение (displacement)
`add EAX, [ESP]`	`EAX <- EAX + Mem[ESP]`	Базовая адресация
`add EAX, [EDX+40]`	`EAX <- EAX + Mem[EDX+40]`	Базовая адресация + смещение
`add EAX, [60+EDI*4]`	`EAX <- EAX + Mem[60+EDI*4]`	Смещение + масштабируемый индекс
`add EAX, [EDX+80+EDI*2]`	`EAX <- EAX + Mem[EDX+80+EDI*2]`	Базовая адресация + смещение + масштабируемый индекс

Таблица 6.12 Инструкции, использующие 8-, 16- или 32-битные операнды

Пример	Назначение	Размер операндов
`add AH, BL`	`AH <- AH + BL`	8 бит
`add AX, -1`	`AX <- AX + 0xFFFF`	16 бит
`add EAX, EDX`	`EAX <- EAX + EDX`	32 бита

6.8.3. Флаги состояния

Как и большинство архитектур CISC, x86 использует флаги состояния (также называемые кодами условий) для принятия решений о переходах и отслеживания переносов и арифметических переполнений. В архитектуре x86 используется 32-битный регистр `EFLAGS`, в котором хранятся флаги состояния. Назначение некоторых битов из регистра `EFLAGS` приведено в **табл. 6.13**. Оставшиеся биты используются операционной системой. Архитектурное состояние процессора x86 включает в себя `EFLAGS`, а также восемь регистров и `EIP`.

Таблица 6.13 Некоторые биты регистра `EFLAGS`

Название	Назначение
`CF` (Carry Flag, флаг переноса)	Показывает, что при выполнении последней арифметической операции результат вышел за пределы разрядной сетки. Указывает на то, что произошло переполнение при беззнаковых вычислениях. Также используется как флаг переноса при работе с числами, разрядность которых превышает разрядность архитектуры
`ZF` (Zero Flag, флаг нуля)	Показывает, что результат последней операции равен нулю
`SF` (Sign Flag, флаг знака)	Показывает, что результат последней операции был отрицательным (старший бит результата равен 1)
`OF` (Overflow Flag, флаг переполнения)	Показывает, что произошло переполнение при вычислениях со знаковыми числами в дополнительном коде

6.8.4. Команды x86

Архитектура x86 имеет большую, чем у RISC-V, систему команд. В **табл. 6.14** показаны некоторые команды общего назначения. Система команд x86 также включает команды обработки чисел с плавающей запятой и коротких векторов упакованных данных. Операнд-назначение обозначен в таблице как D (регистр или ячейка памяти), а операнд-источник обозначен как S (регистр, непосредственный операнд или ячейка памяти).

Обратите внимание, что некоторые команды всегда производят действия только с определенными регистрами. Например, умножение двух 32-битных чисел всегда использует в качестве одного из источников `EAX`

и всегда записывает 64-битный результат в EDX и EAX. Команда LOOP всегда хранит счетчик итераций цикла в ECX, а команды PUSH, POP, CALL и RET используют указатель вершины стека ESP.

Команды условного перехода проверяют значения флагов и, если выполнено соответствующее условие, осуществляют переход. Эти команды имеют много разновидностей. Например, команда JZ осуществляет переход в том случае, когда флаг нуля (ZF) равен 1, а команда JNZ – когда ZF равен 0. Команды перехода обычно следуют за командами, которые устанавливают флаги, такими как команда сравнения (CMP). В **табл. 6.15** перечислены некоторые команды условных переходов и то, как на них воздействуют флаги, предварительно установленные командами сравнения.

Таблица 6.14 Некоторые инструкции x86

Инструкция	Назначение	Функция
ADD/SUB	Сложение/вычитание	D = D + S / D = D – S
ADDC	Сложение с переносом	D = D + S + CF
INC/DEC	Увеличение/уменьшение на единицу	D = D + 1 / D = D – 1
CMP	Сравнение	Установить флаги по результатам D – S
NEG	Инверсия	D = –D
AND/OR/XOR	Логическое И/ИЛИ/Исключающее ИЛИ	D = D операция S
NOT	Логическое НЕ	$D = \overline{D}$
IMUL/MUL	Знаковое/беззнаковое умножение	EDX:EAX = EAX × D
IDIV/DIV	Знаковое/беззнаковое деление	EDX:EAX/D EAX = частное; EDX = остаток
SAR/SHR	Арифметический/логический сдвиг вправо	D = D >>> S / D = D >> S
SAL/SHL	Сдвиг влево	D = D << S
ROR/ROL	Циклический сдвиг вправо/влево	Циклически сдвинуть D на S разрядов
RCR/RCL	Циклический сдвиг вправо/влево через бит переноса	Циклически сдвинуть CF и D на S разрядов
BT	Проверка бита	CF = D[S] (бит номер S из D)
BTR/BTS	Проверить бит и сбросить/установить его	CF = D[S]; D[S] = 0 / 1
TEST	Установить флаги в результате проверки битов	Установить флаги по результатам D AND S
MOV	Скопировать операнд	D = S
PUSH	Поместить в стек	ESP = ESP – 4; Mem[ESP] = S
POP	Прочитать из стека	D = MEM[ESP]; ESP = ESP + 4

Таблица 6.14 (окончание)

Инструкция	Назначение	Функция
CLC, STC	Сбросить/установить флаг переноса	CF = 0 / 1
JMP	Безусловный переход	Переход по относительному адресу: EIP = EIP + S Переход по абсолютному адресу: EIP = S
Jcc	Ветвление (условный переход)	Если установлен флаг, то EIP = EIP + S
LOOP	Проверка условия цикла	ECX = ECX - 1 Если ECX ≠ 0, то EIP = EIP + imm
CALL	Вызов функции	ESP = ESP - 4; MEM[ESP] = EIP; EIP = S
RET	Возврат из функции	EIP = MEM[ESP]; ESP = ESP + 4

Таблица 6.15 Некоторые условия переходов

Инструкция	Назначение	Действие после CMP D, S
JZ/JE	Ветвление, если ZF = 1	Ветвление, если D = S
JNZ/JNE	Ветвление, если ZF = 0	Ветвление, если D ≠ S
JGE	Ветвление, если SF = OF	Ветвление, если D ≥ S
JG	Ветвление, если SF = OF и ZF = 0	Ветвление, если D > S
JLE	Ветвление, если SF ≠ OF и ZF = 1	Ветвление, если D ≤ S
JL	Ветвление, если SF ≠ OF	Ветвление, если D < S
JC/JB	Ветвление, если CF = 1	
JNC	Ветвление, если CF = 0	
JO	Ветвление, если OF = 1	
JNO	Ветвление, если OF = 0	
JS	Ветвление, если SF = 1	
JNS	Ветвление, если SF = 0	

6.8.5. Кодировка команд x86

Кодировка команд x86 – это тяжелое наследие десятилетий постепенных изменений. В отличие от RISC-V, где команды всегда имеют длину 32 бита (или 16 в случае сжатых команд), длина команды x86 может составлять от 1 до 15 байт, как показано на **рис. 6.37**[1].

[1] Если использовать все необязательные поля, то можно собрать команду длиной 17 байт, но x86 имеет ограничение на длину корректной команды, равное 15 байт.

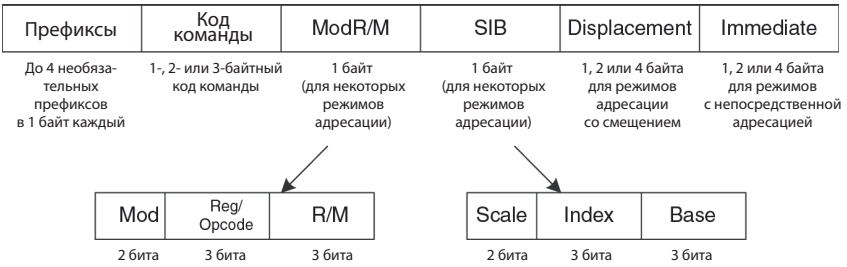


Рис. 6.37 Кодировка команд x86

Код операции (opcode) может составлять 1, 2 или 3 байта. Далее следуют четыре дополнительных поля: `ModR/M`, `SIB`, `Displacement` и `Immediate`. Поле `ModR/M` определяет режим адресации. Поле `SIB` определяет коэффициент масштабирования (scale), индексный (index) и базовый (base) регистры в некоторых режимах адресации. Поле `Displacement` содержит 1-, 2- или 4-байтовое смещение, используемое в соответствующих режимах адресации. Поле `Immediate` содержит 1-, 2- или 4-байтовую константу для команд, использующих непосредственный операнд. Более того, команда может иметь до четырех однобайтных префиксов, изменяющих ее поведение.

Однобайтовое поле `ModR/M` использует 2-битное поле режима `Mod` и 3-битное поле `R/M` для задания режима адресации одного из операндов. Операнд может находиться в одном из восьми регистров, или его можно указать при помощи одного из 24 режимов адресации памяти. Из-за ошибок в кодировке регистры `ESP` и `EBP` не могут использоваться как базовый или индексный регистры в некоторых режимах адресации. В поле `Reg` указывается регистр, используемый в качестве второго операнда. Для некоторых команд, не имеющих второго операнда, поле `Reg` используется для хранения трех дополнительных бит кода операции.

В режимах адресации, использующих регистр масштабируемого индекса, байт `SIB` определяет индексный регистр и коэффициент масштабирования (1, 2, 4 или 8). Если при адресации используются базовый адрес и индекс, то `SIB` также определяет регистр базового адреса.

Архитектура RISC-V позволяет точно определить тип команды по полям кода операции `op`, `funct3` и `funct7`. Архитектура x86 использует разное количество битов для определения разных команд. Часто используемые команды имеют меньший размер, что уменьшает среднюю длину команд в программе. Некоторые команды могут иметь несколько кодов операций. Например, команда `add AL, imm8` выполняет 8-битное сложение регистра `AL` и непосредственного операнда. Эта команда представляется в виде однобайтового кода операции (0x04) и однобайтового непосредственного операнда. Регистр `A` (`AL`, `AX` или `EAX`) называется аккумулятором. С другой стороны, команда `add D, imm8` производит 8-битное сложение непосредственного операнда с операндом `D` и запи-

сывает результат в D, причем D может быть регистром или ячейкой памяти. Эта команда состоит из однобайтового кода операции (0x08), одного или более байтов, определяющих местонахождение D, и однобайтового непосредственного операнда imm8. То есть многие команды имеют более короткие кодировки в том случае, если их результат сохраняется в аккумулятор.

В оригинальном процессоре 8086 в коде операции указывалась разрядность операндов (8 или 16 бит). Когда в процессор 80386 добавили 32-битные операнды, то свободных кодов операции, которые позволили бы добавить новый размер операндов, уже не осталось, поэтому команды, использующие 16-битные и 32-битные операнды, имеют одинаковые коды операции. Чтобы различать их, используют дополнительный бит в *дескрипторе сегмента кода*, который устанавливается операционной системой и указывает процессору, какую команду он должен выполнить. Для обратной совместимости с программами, написанными для 8086, этот бит устанавливается в ноль, после чего все операнды по умолчанию считаются 16-битными. Если же этот бит равен единице, то используются 32-битные операнды. Более того, программист может изменить форму конкретной команды при помощи префикса: если перед кодом операции добавить префикс 0x66, то будет использоваться альтернативный размер операндов (16 бит в 32-битном режиме или 32 бита в 16-битном режиме).

6.8.6. Другие особенности x86

В процессор 80286 был добавлен механизм сегментации для разделения памяти на сегменты размером до 64 Кбайт. Когда операционная система включала сегментацию, то все адреса вычислялись относительно начала сегмента. Процессор проверял адреса и при выходе за пределы сегмента формировал сигнал ошибки, тем самым предотвращая доступ программ за пределы своего сегмента. Сегментация вызывала множество проблем при программировании и в современных версиях операционной системы Windows не используется.

Архитектура x86 поддерживает команды, работающие с цепочками (последовательностями или строками) байтов или слов. Эти команды реализуют операции копирования, сравнения и поиска определенного значения. В современных процессорах такие команды, как правило, работают медленнее, чем последовательность простых команд, делающих то же самое, поэтому их лучше избегать.

Как мы уже упоминали ранее, префикс 0x66 используется для выбора 16-битных или 32-битных операндов. Другие префиксы применяются для захвата внешней шины (это необходимо для обеспечения атомарного доступа к переменным в общей памяти в многопроцессорных системах), предсказания переходов или повторения команды при обработке цепочки байтов или слов.

В середине 1990-х годов Intel и Hewlett-Packard совместно разработали новую 64-битную архитектуру под названием IA-64. Она была разработана с чистого листа, использовала результаты исследований в области компьютерной архитектуры, полученные за 20 лет, прошедших с момента появления 8086, и обеспечивала 64-битное адресное пространство. Тем не менее IA-64 до сих пор не стала успешной на рынке. Большинство компьютеров, которым необходимо большое адресное пространство, используют 64-битные расширения x86.

Проблема любой архитектуры – нехватка памяти. Располагая 32-битными адресами, процессор x86 может получить доступ к 4 ГБ памяти. Это намного больше, чем было у самых больших компьютеров в 1985 году. Но к началу 2000-х годов этого объема памяти перестало хватать. В 2003 году AMD расширила адресное пространство и размеры регистров до 64 бит, выпустив усовершенствованную архитектуру AMD64. Она имеет режим совместимости, который позволяет запускать 32-разрядные программы без изменений, в то время как ОС использует преимущества увеличенного адресного пространства. В 2004 году Intel уступила в конкурентном споре и согласилась принять 64-разрядные расширения, переименовав их в Extended Memory 64 Technology (EM64T). Благодаря 64-битной адресации компьютеры могут получить доступ к 16 эксабайтам (16 млрд ГБ) памяти.

Для читателей, которые заинтересованы в более подробном изучении архитектуры x86, на веб-сайте Intel размещено бесплатное Руководство разработчика программного обеспечения Intel для архитектуры x86[1].

6.8.7. Архитектура x86: подведение итогов

В этом разделе мы рассмотрели основные отличия архитектуры RISC-V от CISC-архитектуры x86. Архитектура x86 позволяет создавать более короткие программы, потому что ее сложные команды эквивалентны нескольким простым командам RISC-V и вдобавок закодированы так, чтобы занимать минимум места в памяти. Но архитектура x86 – это мешанина из всевозможных решений, накопленных за годы разработки. Некоторые из них давно не несут никакой пользы, но приходится сохранять их для обратной совместимости со старыми программами.

У этой архитектуры слишком мало регистров, ее команды сложно декодировать, а набор команд трудно объяснить. Несмотря на эти недостатки, x86 остается доминирующей архитектурой для персональных компьютеров потому, что невозможно переоценить важность совместимости программного обеспечения, и потому, что огромный рынок оправдывает затраты на разработку все более быстрых x86-совместимых микропроцессоров.

---

[1] В настоящее время на сайте Intel доступен обновленный документ Intel® 64 and IA-32 Architectures Software Developer Manuals по адресу https://www.intel.ru/content/www/ru/ru/support/articles/000006715/processors.html. – *Прим. перев.*

6.9. Заключение

Чтобы управлять компьютером, нужно разговаривать на его языке. Архитектура компьютера определяет, как именно нужно это делать. В настоящее время в мире широко используется большое количество разных архитектур, но если вы хорошо поймете одну из них, то изучить остальные будет довольно просто. При изучении новой архитектуры вы должны задать следующие главные вопросы:

- Какова длина слова данных?
- Какие регистры доступны?
- Как организована память?
- Какие есть инструкции?

Архитектура RISC-V (RV32I) является 32-битной потому, что она работает с 32-битными данными. В архитектуре RISC-V определено 32 регистра общего назначения. В принципе, почти любой регистр можно использовать для любой цели. Тем не менее существуют соглашения, по которым определенные регистры зарезервированы для конкретных целей. Это сделано для того, чтобы облегчить процесс программирования, и для того, чтобы функции, разработанные разными программистами, могли легко между собой взаимодействовать. Например, регистр 0 (`zero`) всегда содержит константу 0, регистр `ra` содержит адрес возврата после выполнения инструкции `jal`, а регистры `a0-a7` хранят аргументы функции. Кроме того, регистры `a0` и `a1` хранят возвращаемое значение функции. В архитектуре RISC-V память адресуется побайтово и использует 32-битные адреса. Инструкции имеют длину 32 бита и выровнены в памяти по границе 4-байтного слова для более быстрого доступа к ним. В этой главе мы рассмотрели наиболее часто используемые инструкции RISC-V.

Важность определения компьютерной архитектуры заключается в том, что программа, написанная для выбранной архитектуры, будет работать на совершенно разных реализациях этой архитектуры. Например, программы, написанные для процессора Intel Pentium в 1993 году, будут в общем случае работать (причем работать значительно быстрее) на процессорах Intel Xeon или AMD Phenom в 2022 году.

В первой половине этой книги мы узнали о схемных и логических уровнях абстракции. В этой главе мы перешли на архитектурный уровень. В следующей главе мы изучим микроархитектуру – способ организации цифровых строительных блоков, с помощью которых создается аппаратная реализация архитектуры процессора.

Микроархитектура – это мост между электрическими схемами и программированием. По нашему мнению, изучение микроархитектуры является одним из наиболее захватывающих занятий для инженера: вы узнаете, как создать собственный микропроцессор!

Упражнения

Упражнение 6.1 Приведите три примера из архитектуры RISC-V для каждого из принципов хорошей разработки: (1) для простоты придерживайтесь единообразия; (2) типичный сценарий должен быть быстрым; (3) чем меньше, тем быстрее; (4) хорошая разработка требует хороших компромиссов. Поясните, как каждый из ваших примеров иллюстрирует соответствующий принцип.

Упражнение 6.2 Архитектура RISC-V содержит набор 32-битных регистров. Можно ли создать компьютерную архитектуру без регистров? Если можно, кратко опишите такую архитектуру и ее систему команд. Какие преимущества и недостатки будут у этой архитектуры по сравнению с архитектурой RISC-V?

Упражнение 6.3 Напишите следующие строки, используя кодировку ASCII. Запишите окончательные ответы в шестнадцатеричном формате.

(a) hello there
(b) bag o' chips
(c) To the rescue!

Упражнение 6.4 Повторите упражнение 6.3 для следующих строк:

(a) Cool
(b) RISC-V
(c) boo!

Упражнение 6.5 Покажите, как строки из **упражнения 6.3** хранятся побайтово в адресуемой памяти, начиная с адреса памяти 0x004F05BC. Первый символ строки сохраняется по младшему байтовому адресу (в данном случае 0x004F05BC). Укажите в явном виде адрес каждого байта в памяти.

Упражнение 6.6 Повторите **упражнение 6.5** для строк из **упражнения 6.4**.

Упражнение 6.7 Инструкция `nor` не входит в набор инструкций RISC-V, потому что эквивалентная операция может быть реализована с использованием существующих инструкций. Напишите короткий фрагмент ассемблерного кода, который выполняет следующую операцию: `s3 = s4 NOR s5`. Используйте наименьшее возможное число инструкций.

Упражнение 6.8 Инструкция `nand` не входит в набор инструкций RISC-V, потому что эквивалентная операция может быть реализована с использованием существующих инструкций. Напишите короткий фрагмент ассемблерного кода, который выполняет следующую операцию: `s3 = s4 NAND s5`. Используйте наименьшее возможное число инструкций.

Упражнение 6.9 Преобразуйте следующие фрагменты кода на языке высокого уровня в язык ассемблера RISC-V. Предположим, что знаковые целочисленные переменные `g` и `h` хранятся в регистрах `a0` и `a1` соответственно. Снабдите свой код подробными комментариями.

(a)
```
if (g > h)
  g = g + 1;
else
  h = h - 1;
```

(b)
```
if (g <= h)
  g = 0;
else
  h = 0;
```

Упражнение 6.10 Повторите **упражнение 6.9** для следующих фрагментов кода:

(a)
```
if (g >= h)
  g = g + h;
else
  g = g - h;
```

(b)
```
if (g < h)
  h = h + 1;
else
  h = h * 2;
```

Упражнение 6.11 Преобразуйте следующий фрагмент кода на языке высокого уровня в язык ассемблера RISC-V. Предположим, что базовые адреса `array1` и `array2` хранятся в регистрах `t1` и `t2` соответственно и что массив `array2` уже инициализирован перед использованием. Используйте наименьшее возможное число инструкций. Снабдите свой код подробными комментариями.

```
int i;
int array1[100];
int array2[100];
...
for (i = 0; i < 100; i = i + 1)
  array1[i] = array2[i];
```

Упражнение 6.12 Повторите **упражнение 6.11** для следующего фрагмента кода на языке высокого уровня. Предположим, что временный массив уже проинициализирован перед использованием и что `t3` содержит базовый адрес `temp`.

```
int i;
int temp[100];
...
for (i = 0; i < 100; i = i + 1)
  temp[i] = temp[i] * 128;
```

Упражнение 6.13 Разработайте ассемблерный код RISC-V для сохранения следующих констант в регистре `s7`. Используйте наименьшее возможное число инструкций.

(a) 29
(b) −214
(c) −2999
(d) 0xABCDE000
(e) 0xEDCBA123
(f) 0xEEEEEFAB

Упражнение 6.14 Повторите **упражнение 6.13** для следующих констант:

(a) 47
(b) −349
(c) 5328

(d) 0xBBCCD000
(e) 0xFEEBC789
(f) 0xCCAAB9AB

Упражнение 6.15 Разработайте функцию на языке высокого уровня (например, C), имеющую следующий вид: `int find42(int array[], int size)`. Здесь `array` задает базовый адрес некоторого массива целых чисел, а `size` содержит число элементов в этом массиве. Функция должна возвращать порядковый номер первого элемента массива, содержащего значение 42. Если в массиве нет числа 42, то функция должна вернуть −1. Снабдите свой код подробными комментариями.

Упражнение 6.16 Функция на языке высокого уровня `strcpy` (string copy, копирование строки) копирует символьную строку `src` в символьную строку `dst`.

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Реализуйте приведенную выше функцию `strcpy` на языке ассемблера RISC-V. Используйте регистр `s0` для `i`.
(b) Изобразите стек до вызова, во время и после вызова функции `strcpy`. Считайте, что перед вызовом `strcpy` значение `sp` = 0xFFC000.

Эта простая функция копирования строки имеет один весьма серьезный недостаток: она не может узнать, зарезервировано ли достаточно места в памяти по адресу `dst`, чтобы скопировать туда исходную строку. Если компьютерный взломщик может заставить программу выполнить функцию `strcpy` с чрезмерно длинной строкой, находящейся по адресу `src`, то `strcpy` может изменить важные данные и даже инструкции в памяти программы, располагающиеся за зарезервированным участком памяти. Ловко модифицированный код может «захватить» компьютер и подчинить его действия взломщику. Это так называемая *атака переполнения буфера*. Она используется вредоносными программами, в частности печально известным «червем» Blaster, который причинил ущерб приблизительно на 525 млн долларов в 2003 году.

Упражнение 6.17 Преобразуйте функцию на языке высокого уровня из **упражнения 6.15** в ассемблерный код RISC-V. Снабдите свой код подробными комментариями.

Упражнение 6.18 Рассмотрим приведенный ниже код на языке ассемблера RISC-V. Функции `func1`, `func2` и `func3` – нелистовые функции, а `func4` – листовая. Полный код функций не показан, но в комментариях указаны регистры, используемые каждой из них. Предположим, что функциям не нужно сохранять какие-либо необеременяемые регистры в своих стеках.

```
0x00091000 func1: ... # func1 использует t2–t3, s4–s10
...
0x00091020 jal func2
...
0x00091100 func2: ... # func2 использует a0–a2, s0–s5
...
0x0009117C jal func3
...
0x00091400 func3: ... # func3 использует t3, s7–s9
...
0x00091704 jal func4
...
```

```
0x00093008 func4: ... # func4 использует s10–s12
...
0x00093118 jr ra
```

a) Сколько слов занимает фрейм стека у каждой из этих функций?
b) Изобразите стек после вызова `func4`. Укажите, какие регистры хранятся в стеке и где именно. Отметьте каждый из фреймов стека. Там, где это возможно, подпишите значения, сохраненные в стеке. Предположим, что `sp` = 0xABC124 непосредственно перед вызовом `func1`.

Упражнение 6.19 Каждое число в последовательности Фибоначчи является суммой двух предыдущих чисел. В **табл. 6.16** перечислены первые числа последовательности $fib(n)$.

Таблица 6.16 Последовательность Фибоначчи

n	1	2	3	4	5	6	7	8	9	10	11	...
$fib(n)$	1	1	2	3	5	8	13	21	34	55	89	...

(a) Чему равны значения $fib(n)$ для $n = 0$ и $n = -1$?
(b) Напишите функцию с именем `fib` на языке высокого уровня. Функция должна возвращать число Фибоначчи для любого неотрицательного значения n. *Подсказка*: используйте цикл. Прокомментируйте ваш код.
(c) Преобразуйте функцию, разработанную в части (b), в код на ассемблере RISC-V. После каждой строки кода добавьте строку комментария, поясняющего, что она делает. Проведите тестирование вполнения кода для случая $fib(9)$ в симуляторе RISC-V (чтобы узнать, как установить симулятор RISC-V, обратитесь к **предисловию**.)

Упражнение 6.20 Проанализируйте **пример кода 6.28**. В этом упражнении предположим, что функция `factorial(n)` вызывается с аргументом `n` = 5.

(a) Чему будет равен регистр `a0`, когда функция `factorial` завершится и управление будет возвращено вызвавшей ее функции?
(b) Предположим, вы заменили инструкции по адресам 0x8508 и 0x852C на `nop`. Как будет вести себя программа: (1) войдет в бесконечный цикл, но не завершится аварийно; (2) завершится аварийно (произойдет переполнение стека или счетчик команд выйдет за пределы программы); (3) вернет неправильное значение в `a0`, когда программа вернется в цикл (если да, то какое значение?); (4) продолжит работать правильно, несмотря на изменения?
(c) Повторите часть (b) со следующими изменениями кода:
  (i) замените инструкции по адресам 0x8504 и 0x8528 на `nop`;
  (ii) замените инструкцию по адресу 0x8518 на `nop`;
  (iii) замените инструкцию по адресу 0x8530 на `nop`.

Упражнение 6.21 Бен Битдидл попытался вычислить функцию $f(a,b) = 2a + 3b$ для положительного значения b, но переусердствовал с вызовами функций и рекурсией и разработал вот такой код для функций `f` и `g`:

```
// код на языке высокого уровня для функций f и g
int f(int a, int b) {
```

```
  int j;

  j = a;
  return j + a + g(b);
}
int g(int x) {
  int k;

  k = 3;
  if (x = = 0) return 0;
  else return k + g(x - 1);
}
```

После этого Бен транслировал эти две функции на язык ассемблера RISC-V. Он также разработал функцию `test`, которая вызывает функцию *f*(5,3).

```
# код на языке ассемблера RISC-V
# f: a0 = a, a1 = b, s4 = j;
# g: a0 = x, s4 = k
0x8000 test:  addi a0, zero, 5    # a = 5
0x8004        addi a1, zero, 3    # b = 3
0x8008        jal  f              # вызов f(5, 3)
0x800C loop:  j    loop           # вечный цикл
0x8010 f:     addi sp, sp, -16    # создать фрейм в стеке
0x8014        sw   a0, 0xC(sp)    # сохранить a0
0x8018        sw   a1, 0x8(sp)    # сохранить a1
0x801C        sw   ra, 0x4(sp)    # сохранить ra
0x8020        sw   s4, 0x0(sp)    # сохранить s4
0x8024        addi s4, a0, 0      # j = a
0x8028        addi a0, a1, 0      # поместить b как аргумент для g()
0x802C        jal  g              # вызов g
0x8030        lw   t0, 0xC(sp)    # восстановить a в t0
0x8034        add  a0, a0, t0     # a0 = g(b) + a
0x8038        add  a0, a0, s4     # a0 = (g(b) + a) + j
0x803C        lw   s4, 0x0(sp)    # восстановить регистры
0x8040        lw   ra, 0x4(sp)
0x8044        addi sp, sp, 16
0x8048        jr   ra             # возврат в точку вызова
0x804C g:     addi sp, sp, -8     # создать фрейм в стеке
0x8050        sw   ra, 4(sp)      # сохранить регистры
0x8054        sw   s4, 0(sp)
0x8058        addi s4, zero, 3    # k = 3
0x805C        bne  a0, zero, else # если (x != 0), перейти к метке else
0x8060        addi a0, zero, 0    # возвращаемое значение 0
0x8064        j    done           # очистка и возврат
0x8068 else:  addi a0, a0, -1     # уменьшить x на 1
0x806C        jal  g              # вызов g(x - 1)
0x8070        add  a0, s4, a0     # возвращаемое значение k + g(x - 1)
0x8074 done:  lw   s4, 0(sp)      # восстановить регистры
0x8078        lw   ra, 4(sp)
0x807C        addi sp, sp, 8
0x8080        jr   ra             # возврат в точку вызова
```

Вам может быть полезно изобразить стек по примеру **рис. 6.10**, чтобы ответить на следующие вопросы:

(a) если код выполнится, начиная с метки `test`, то какое значение окажется в регистре `a0`, когда программа дойдет до метки `loop`? Правильно ли программа вычислит 2a + 3b?

(b) предположим, Бен заменил инструкцию по адресу 0x8014 на `nop`. В этом случае программа (1) войдет в бесконечный цикл, но не остановится; (2) завершится аварийно (произойдет переполнение стека или счетчик команд выйдет за пределы программы); (3) вернет неправильное значение в `a0`, когда возвратится в цикл (если да, то какое значение?); (4) будет работать правильно, несмотря на изменения?

(c) Повторите часть (b), когда будут удалены (заменены на `nop`) инструкции по следующим адресам. Обратите внимание, что удаляются только инструкции, но не метки:
  (i) 0x8014 и 0x8030
  (ii) 0x803C и 0x8040
  (iii) 0x803C
  (iv) 0x8030
  (v) 0x8054 и 0x8074
  (vi) 0x8020 и 0x803C
  (vii) 0x8050 и 0x8078.

Упражнение 6.22 Преобразуйте следующий ассемблерный код RISC-V в машинный код. Запишите команды в шестнадцатеричном формате.

```
addi s3, s4, 28
sll  t1, t2, t3
srli s3, s1, 14
sw   s9, 16(t4)
```

Упражнение 6.23 Повторите **упражнение 6.22** для следующего ассемблерного кода RISC-V:

```
add  s7, s8, s9
srai t0, t1, 0xC
ori  s3, s1, -1348
lw   s4, 0x5C(t3)
```

Упражнение 6.24 Предположим, что нас интересуют только команды с полем константы.

(a) Какие команды из **упражнения 6.22** содержат поле константы, будучи представленными в формате машинного кода?

(b) К какому типу (*I*, *S*, *B*, *U* или *J*) относятся инструкции из части (a)?

(c) Запишите 5–21-битные значения полей констант каждой команды из части (a) в шестнадцатеричном формате. Если значения дополнены, также сразу запишите их в 32-битном формате. В противном случае укажите, что они не дополняются.

Упражнение 6.25 Повторите задание из **упражнения 6.24** для инструкций из **упражнения 6.23**.

Упражнение 6.26 Рассмотрим приведенный ниже фрагмент машинного кода RISC-V. Первая инструкция указана вверху.

(a) Преобразуйте фрагмент машинного кода в язык ассемблера RISC-V.

(b) Путем обратного инжиниринга получите исходный код программы на языке высокого уровня, которая компилируется в функцию на языке ассемблера из пункта (a). Снабдите свой код подробными комментариями.

(c) Кратко запишите словами, что делает программа. Регистры `a0` и `a1` – это входные параметры (аргументы функции), и они изначально содержат положительные числа A и B. В конце программы регистр `a0` хранит возвращаемое значение.

```
0x01800513
0x00300593
0x00000393
0x00058E33
0x01C54863
0x00138393
0x00BE0E33
0xFF5FF06F
0x00038533
```

Упражнение 6.27 Повторите **упражнение 6.26** для следующего машинного кода. Регистры `a0` и `a1` содержат входные параметры. Регистр `a0` содержит 32-битное число, а регистр `a1` – адрес 32-элементного массива символов (`char`).

```
0x01F00393
0x00755E33
0x001E7E13
0x01C580A3
0x00158593
0xFFF38393
0xFE03D6E3
0x00008067
```

Упражнение 6.28 Переведите приведенные ниже инструкции условного перехода в машинный код. Адреса инструкций указаны слева от каждой из них:

(a)
```
0x0000A000       beq t4, zero, Loop
0x0000A004       ...
0x0000A008       ...
0x0000A00C Loop: ...
```

(b)
```
0x00801000       bne s5, a1, L1
...              ...
0x0080174C L1:   ...
```

(c)
```
0x0000C10C Back: ...
...              ...
0x0000D000       blt s1, s2, Back
```

(d)
```
0x01030AAC       bge t4, t6, L2
...              ...
0x01031AA4 L2:   ...
```

(e)
```
0x0BC08004 L3:   ...
...              ...
0x0BC09000       beq s3, s7, L3
```

Упражнение 6.29 Переведите приведенные ниже инструкции условного перехода в машинный код. Адреса инструкций указаны слева от каждой из них:

(a)
```
0xAA00E124       blt t4, s3, Loop
0xAA00E128       ...
0xAA00E12C       ...
0xAA00E130 Loop: ...
```

(b)
```
0xC0901000      bge t1, t2, L1
...             ...
0xC090174C L1:  ...
```

(c)
```
0x1230D10C Back: ...
...              ...
0x1230D908       bne s10, s11, Back
```

(d)
```
0xAB0C99A8      beq a0, s1, L2
...             ...
0xAB0CA0FC L2:  ...
```

(e)
```
0xFFABCF04 L3:  ...
...             ...
0xFFABD640      blt s1, t3, L3
```

Упражнение 6.30 Переведите приведенные ниже инструкции условного перехода в машинный код. Адреса инструкций указаны слева от каждой из них:

(a)
```
0x1234ABC0      j Loop
...             ...
0x123CABBC Loop: ...
```

(b)
```
0x12345678 Back: ...
...              ...
0x123B8760       jal s0, Back
```

(c)
```
0xAABBCCD0      jal L1
...             ...
0xAABDCD98 L1:  ...
```

(d)
```
0x11223344      j L2
...             ...
0x1127BCDC L2:  ...
```

(e)
```
0x9876543C L3:  ...
...             ...
0x9886543C      jal L3
```

Упражнение 6.31 Переведите приведенные ниже инструкции условного перехода в машинный код. Адреса инструкций указаны слева от каждой из них:

(a)
```
0x0000ABC0      jal Loop
...             ...
0x0000EEEC Loop: ...
```

(b)
```
0x0000C10C Back: ...
...              ...
0x000F1230       jal Back
```

(c)
```
0x00801000      jal s1, L1
...             ...
0x008FFFDC L1:  ...
```

(d)
```
0xA1234560      j L2
...             ...
0xA131347C L2:  ...
```

(e)
```
0xF0BBCCD4 L3:  ...
...             ...
0xF0CBCCD4      j L3
```

Упражнение 6.32 Рассмотрим следующий фрагмент кода на языке ассемблера RISC-V. Числа слева от каждой инструкции указывают ее адрес:

```
0xA0028 Func1: addi t4, a1, 0
0xA002C        ori  a0, a0, 32
0xA0030        sub  a1, a1, a0
0xA0034        jal  Func2
...            ...
0xA0058 Func2: lw   t2, 4(a0)
0xA005C        sw   t2, 16(a1)
0xA0060        srli t3, t2, 8
0xA0064        beq  t2, t3, Else
0xA0068        jr   ra
0xA006C Else:  addi a0, a0, 4
0xA0070        j    Func2
```

(a) Преобразуйте последовательность инструкций в машинный код в шестнадцатеричном формате.

(b) Сделайте список типов инструкций и режимов адресации, которые были использованы для каждой строки кода.

Упражнение 6.33 Рассмотрим следующий фрагмент кода на C:

```
// код на C
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1)
    array[i] = compare(num, i);
}

int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
  else
    return 0;
}
int sub(int a, int b) {
    return a - b;
}
```

(a) Запишите этот фрагмент кода на языке ассемблера RISC-V. Используйте регистр `s4` для хранения переменной `i`. Следите за тем, чтобы правильно работать с указателем стека. Массив хранится в стеке функции `setArray` (рассмотрено в конце **раздела 6.3.7**). Снабдите свой код подробными комментариями.

(b) Предположим, что первой вызванной функцией будет `setArray`. Нарисуйте состояние стека перед вызовом `setArray` и во время каждого последующего вызова. Укажите имена регистров и переменных, хранящихся в стеке. Отметьте расположение указателя `sp` и каждого фрейма стека. Предположим, что изначально `sp` указывает на 0x8000.

(c) Как бы работал ваш код, если бы вы забыли сохранить в стеке регистр `ra`?

Упражнение 6.34 Рассмотрим следующий фрагмент кода на C:

```
// код на C
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n = = 0)
    b = 10;
  else
    b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

(a) Преобразуйте функцию `f` на язык ассемблера RISC-V. Обратите особое внимание на правильность сохранения и восстановления регистров между вызовами функций, а также на использование соглашений о сохранении регистров RISC-V. Предположим, что функция начинается с адреса 0x8100. Храните локальную переменную `b` в регистре `s4`. Снабдите свой код подробными комментариями.

(b) Пошагово вручную выполните функцию из пункта (a) для случая `f(2, 4)`. Изобразите стек, как на **рис. 6.10**, предполагая, что в момент вызова `f` значение `sp` равно 0xBFF00100. Запишите адреса стека, имена регистров и значения данных, хранящиеся в каждом фрейме стека, и опишите, как будет меняться значение указателя стека `sp`. Четко обозначьте каждый фрейм стека. Вам также может быть полезно отслеживать значения в `a0`, `a1` и `s4` в процессе выполнения программы. Предположим, что при вызове `f` значение `s4` = 0xABCD и `ra` = 0x8010.

(c) Каким будет конечный результат в регистре `a0` при вызове `f(2, 4)`?

Упражнение 6.35 Каков максимальный диапазон адресов, по которым инструкции условного перехода (например, `beq`) могут переходить вперед (т. е. в сторону более высоких адресов инструкций)?

Упражнение 6.36 Каков максимальный диапазон адресов, по которым инструкции условного перехода (например, `beq`) могут переходить назад (т. е. в сторону более низких адресов инструкций)?

Упражнение 6.37 Напишите код на языке ассемблера, который выполняет условный переход к инструкции, отстоящей на 32 мегаинструкции впереди от текущей инструкции. Напомним, что 1 мегаинструкция = 2^{20} инструкций = 1 048 576 инструкций. Предположим, что ваш код начинается с адреса 0x8000. Используйте минимально возможное количество инструкций.

Упражнение 6.38 Объясните, почему выгодно иметь большое поле константы в машинном формате инструкции безусловного перехода `jal`.

Упражнение 6.39 Рассмотрим функцию, которая получает массив из 10 элементов 32-битных целых чисел, хранящихся в формате с прямым порядком следования байтов (от младшего к старшему, little-endian), и преобразует его в формат с обратным порядком байтов (от старшего к младшему, big-endian).

(a) Напишите эту функцию на языке высокого уровня.

(b) Перепишите эту функцию на языке ассемблера RISC-V. Тщательно прокомментируйте весь код. Используйте минимально возможное количество инструкций.

Упражнение 6.40 Рассмотрим две строки: `string1` и `string2`.

(a) Напишите код на языке высокого уровня для функции под названием `concat`, которая соединяет их (склеивает вместе): `void concat(char string1[], char string2[], char stringconcat[])`. Заметьте, что эта функция не возвращает значения (т. е. тип возвращаемого значения равен `void`). Результат объединения `string1` и `string2` помещается в строку в `stringconcat`. Предполагается, что массив символов `stringconcat` является достаточно большим, чтобы вместить результат.

(b) Запишите код из части (a) на языке ассемблера RISC-V. Тщательно прокомментируйте весь код.

Упражнение 6.41 Разработайте программу на языке ассемблера RISC-V, которая складывает два положительных числа с плавающей запятой одинарной точности, сохраненных в регистрах `a0` и `a1`. Не используйте инструкции RISC-V для работы с плавающей запятой. В этом упражнении вам не нужно беспокоиться о кодах значений, зарезервированных для специальных целей (например, 0, NaN и т. д.), а также о возможных переполнениях или потере точности. Воспользуйтесь симулятором, чтобы проверить свой код (про симулятор RISC-V рассказано в **предисловии**). Вам нужно будет вручную установить значения `a0` и `a1`, чтобы провести тестирование кода. Продемонстрируйте, что ваш код работает надежно. Снабдите код подробными комментариями.

Упражнение 6.42 Дополните код на языке ассемблера RISC-V из **упражнения 6.41**, чтобы он мог обрабатывать как положительные, так и отрицательные числа с плавающей запятой одинарной точности. Снабдите код подробными комментариями.

Упражнение 6.43 Рассмотрим функцию, которая сортирует значения 10-элементного массива `scores` от наименьшего к наибольшему. После завершения выполнения функции элемент `scores[0]` содержит наименьшее значение, а `scores[9]` – наибольшее значение.

(a) Напишите на языке высокого уровня функцию `sort`, которая выполняет указанную выше задачу. Функция `sort` получает единственный аргумент – адрес массива `scores`. Снабдите код подробными комментариями.

(b) Перепишите функцию `sort` на языке ассемблера RISC-V. Снабдите код подробными комментариями.

Упражнение 6.44 Рассмотрим представленную ниже программу на языке ассемблера RISC-V. Предположим, что инструкции размещены в памяти начиная с адреса 0x8400 и что глобальные переменные `x` и `y` находятся по адресам памяти 0x10024 и 0x10028 соответственно.

```
# код на языке ассемблера RISC-V
main:
  addi sp, sp, -4   # выделить место в стеке
```

```
  sw   ra, 0(sp)    # сохранить ra в стек
  lw   a0, -940(gp) # a0 = x
  lw   a1, -936(gp) # a1 = y
  jal  diff         # вызов diff()
  lw   ra, 0(sp)    # восстановить значение регистров
  addi sp, sp, 4
  jr   ra           # возврат результата
diff:
  sub  a0, a0, a1   # возврат с результатом (a0-a1)
  jr   ra
```

(a) Укажите адрес в памяти рядом с каждой инструкцией ассемблера.
(b) Составьте таблицу символов: укажите имя, адрес и размер каждого символа (т. е. метку функции и глобальную переменную).
(c) Преобразуйте все ассемблерные инструкции в машинный код RISC-V.
(d) Укажите размеры в байтах сегментов данных и кода.
(e) Нарисуйте карту памяти, аналогичную **рис. 6.34**, и покажите на ней места, где хранятся данные и команды. Обязательно укажите значения PC и gp в начале программы.

Упражнение 6.45 Повторите **упражнение 6.44** для представленной ниже программы на языке ассемблера RISC-V. Предположим, что инструкции размещены в памяти начиная с адреса 0x8534 и что глобальные переменные g и h находятся по адресам памяти 0x1305C и 0x13060 соответственно.

```
# код на языке ассемблера RISC-V
main:      addi sp, sp, -8
           sw ra, 4(sp)
           sw s4, 0(sp)
           addi s4, zero, 15
           sw s4, -300(gp)   # g = 15
           addi a1, zero, 27 # arg1 = 27
           sw a1, -296(gp)   # h = 27
           lw a0, -300(gp)   # arg0 = g = 15
           jal greater
           lw s4, 0(sp)
           lw ra, 4(sp)
           addi sp, sp, 8
           jr ra
greater:   blt a1, a0, isGreater
           addi a0, zero, 0
           jr ra
isGreater: addi a0, zero, 1
           jr ra
```

Упражнение 6.46 Объясните преимущества и недостатки хаотичного чередования битов констант, присущего двоичному машинному коду RISC-V.

Упражнение 6.47 Как вам уже известно, в инструкциях RISC-V константы могут дополняться знаковым битом. Спроектируйте блок дополнения констант знаковым битом для RISC-V, используя следующие шаги. Сведите к минимуму необходимое оборудование.

(a) Нарисуйте схему блока дополнения знаковым битом, который расширяет 12-битные константы в инструкциях типа *I*. Вход схемы – это старшие 12 бит инструкции $Instr_{31:20}$, которая содержит 12-битную констан-

ту со знаком. На выходе схемы вы должны получить дополненную до 32 бит константу со знаком $ImmExt_{31:0}$.

(b) Используйте блок дополнения знаковым битом из части (a), чтобы аналогичным образом расширить 12-битную константу, представленную в инструкции типа *S*. При необходимости измените входные данные. Постарайтесь повторно использовать оборудование.

(c) Используйте блок дополнения знаковым битом из части (b), чтобы аналогичным образом расширить 13-битные константы со знаком, представленные в инструкциях типа *B*.

(d) Используйте блок дополнения знаковым битом из части (c), чтобы аналогичным образом расширить 21-битные константы со знаком, представленные в инструкциях типа *J*.

Упражнение 6.48 В этом упражнении необходимо разработать альтернативный блок дополнения констант для RISC-V, используя минимально необходимое оборудование. Предположим, что архитекторы RISC-V решили использовать более понятную людям систему кодирования констант, как показано **на рис. 6.38**. На этом рисунке даны все поля команд, кроме кода операции `op`. В кодировке, представленной на рисунке, не применяется чередование битов (но в командах типа *S* / *B* константа все же разбивается на два поля). Биты, которые отличаются от фактического кода команд в архитектуре RISC-V (показанного на **рис. 6.27**), выделены синим цветом. В частности, эта гипотетическая упрощенная кодировка отличается от фактической кодировки констант в архитектуре RISC-V для форматов типа *B* и *J*.

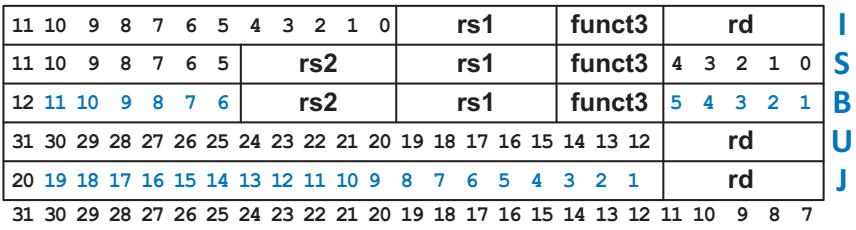


Рис. 6.38 Альтернативная кодировка констант

(a) Нарисуйте схему блока дополнения знаковым битом, который расширяет 12-битные константы в инструкциях типа *I*. Вход схемы – это старшие 12 бит инструкции $Instr_{31:20}$, которая содержит 12-битную константу со знаком. На выходе схемы вы должны получить дополненную до 32 бит константу со знаком $ImmExt_{31:0}$.

(b) Используйте блок дополнения знаковым битом из части (a), чтобы аналогичным образом расширить 12-битную константу, представленную в инструкции типа *S*. При необходимости измените входные данные. Постарайтесь повторно использовать оборудование.

(c) Используйте блок дополнения знаковым битом из части (b), чтобы аналогичным образом расширить 13-битные константы со знаком, представленные в *модифицированных* инструкциях типа *B* (**рис. 6.38**).

(d) Используйте блок дополнения знаковым битом из части (c), чтобы аналогичным образом расширить 21-битные константы со знаком, представленные в *модифицированных* инструкциях типа *J* (**рис. 6.38**).

(e) Если вы выполнили **упражнение 6.47**, сравните свое решение с реализацией в настоящем модуле дополнения RISC-V.

Упражнение 6.49 Подумайте, насколько далеко могут совершать переход инструкции `jal`.

(a) На сколько инструкций может перейти вперед инструкция `jal` (т. е. в сторону более высоких адресов)?
(b) На сколько инструкций может перейти назад инструкция `jal` (т. е. в сторону более низких адресов)?

Упражнение 6.50 Рассмотрим 32-битное слово, хранящееся в 42-м слове памяти с побайтовой адресацией. Напомним, что нулевое слово хранится по адресу памяти 0, первое слово – по адресу 4 и т. д.

(a) Каков байтовый адрес 42-го слова, хранящегося в памяти?
(b) Какие байтовые адреса занимает 42-е слово?
(c) Нарисуйте схему размещения в памяти числа 0xFF223344, хранящегося в слове 42 в машинах с прямым и обратным порядками байтов. Точно обозначьте байтовый адрес, соответствующий каждому значению байта данных.

Упражнение 6.51 Повторите **упражнение 6.50** для 32-битного слова, хранящегося в 15-м слове памяти с побайтовой адресацией.

Упражнение 6.52 Объясните, как следующую программу на языке ассемблера RISC-V можно использовать для определения того, какой порядок байтов использует компьютер – прямой или обратный:

```
addi s7, 100
lui  s3, 0xABCD8   # s3 = 0xABCD8000
addi s3, s3, 0x765 # s3 = 0xABCD8765
sw   s3, 0(s7)
lb   s2, 1(s7)
```

Вопросы для собеседования

Приведенные ниже вопросы обычно задают на собеседованиях на вакансии разработчиков цифровой аппаратуры (но эти вопросы относятся и к любым языкам ассемблера).

Вопрос 6.1 Напишите программу на языке ассемблера RISC-V, которая меняет местами содержимое двух регистров, `a0` и `a1`. Программа не должна использовать другие регистры.

Вопрос 6.2 Предположим, что у вас есть массив из положительных и отрицательных целых чисел. Напишите программу на ассемблере RISC-V, которая находит подмножество массива с максимальной суммой. Адрес массива и количество элементов хранятся в регистрах `a0` и `a1` соответственно. Программа должна поместить найденное подмножество массива, начиная с адреса, находящегося в регистре `a2`. Код должен работать максимально быстро.

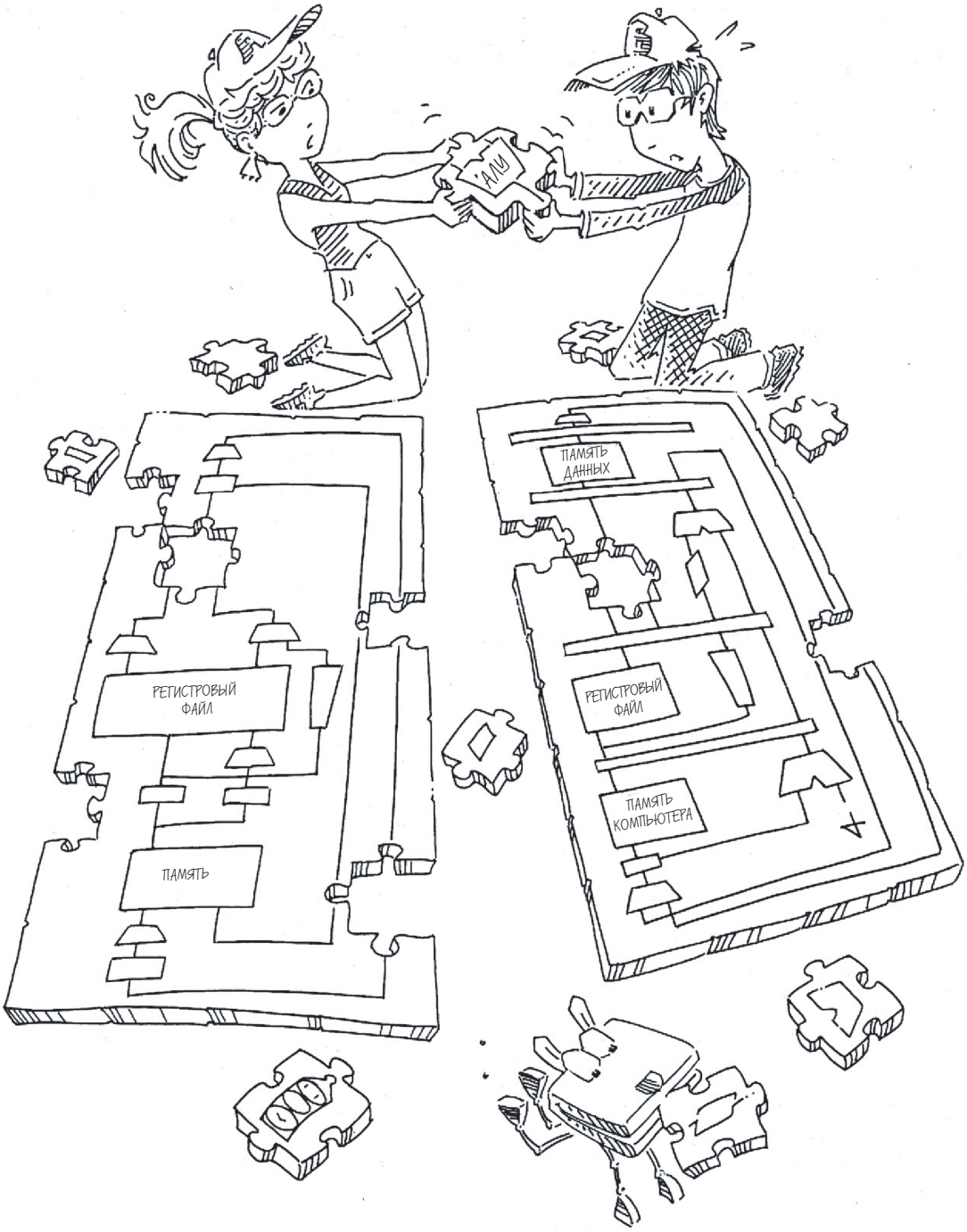
Вопрос 6.3 Дан массив, хранящий строку языка C. Строка содержит предложение. Придумайте алгоритм, который запишет слова в предложении в обратном порядке и сохранит результат обратно в этот массив. Реализуйте ваш алгоритм на языке ассемблера RISC-V.

Вопрос 6.4 Придумайте алгоритм подсчета количества единиц в 32-битном числе. Реализуйте ваш алгоритм на языке ассемблера RISC-V.

Вопрос 6.5 Напишите программу на языке ассемблера RISC-V, меняющую порядок битов в регистре на обратный. Используйте как можно меньше инструкций.

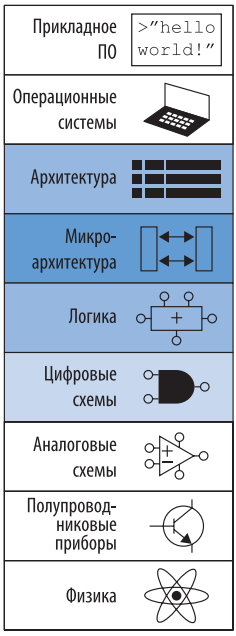
Вопрос 6.6 Напишите программу на языке ассемблера RISC-V, проверяющую, произошло ли переполнение при вычитании значения регистра `a2` из `a3`. Используйте как можно меньше инструкций.

Вопрос 6.7 Придумайте алгоритм, который проверяет, является ли заданная строка палиндромом (*палиндром* – это слово, которое читается в обоих направлениях одинаково, например «wow» или «racecar»). Напишите программу на языке ассемблера RISC-V, реализующую этот алгоритм.


ALU
ПАМЯТЬ ДАННЫХ
РЕГИСТРОВЫЙ ФАЙЛ
ПАМЯТЬ КОМПЬЮТЕРА
РЕГИСТРОВЫЙ ФАЙЛ
ПАМЯТЬ

ГЛАВА 7

Микроархитектура



7.1. Введение

Из этой главы вы узнаете, как собрать собственную версию, а точнее три версии процессора RISC-V, отличающиеся между собой разным соотношением производительности, цены и сложности.

Для непосвященных создание микропроцессора выглядит как волшебство. На самом деле это относительно просто, и, прочитав предыдущие главы этой книги, вы уже знаете все, что нужно. В частности, вы изучили разработку комбинационных и последовательностных схем в соответствии с заданными функциональными и временными ограничениями. Вы познакомились с построением арифметических схем и блоков памяти. Также изучили архитектуру RISC-V, описывающую регистры, команды и память так, как видит их программист.

Эта глава посвящена *микроархитектуре*, которая является связующим звеном между логическими схемами и архитектурой. Микроархитектура описывает, как именно в процессоре расположены и соединены друг с другом регистры, АЛУ, конечные автоматы, блоки памяти и другие блоки, необходимые для реализации архитектуры процессора. У каждой

архитектуры, включая RISC-V, может быть много различных микроархитектур, обеспечивающих разное соотношение производительности, цены и сложности. Все они смогут выполнять одни и те же программы, но их внутреннее устройство может очень сильно отличаться. В этой главе мы разработаем три различные микроархитектуры, чтобы проиллюстрировать компромиссы, на которые приходится идти разработчику.

7.1.1. Архитектурное состояние и система команд

Напомним, что компьютерная архитектура определяется набором команд и архитектурным состоянием. Архитектурное состояние процессора RISC-V определяется содержимым счетчика команд (program counter, PC) и 32 видимых программисту регистров, поэтому любой процессор, реализующий архитектуру RISC-V, вне зависимости от его микроархитектуры обязан иметь счетчик команд и ровно 32 регистра. Зная текущее архитектурное состояние, процессор точно знает, какую операцию над какими данными надо выполнить для получения нового архитектурного состояния. У некоторых микроархитектур есть также и неархитектурное (т. е. не видимое программисту) состояние, которое используется или для упрощения логики, или для улучшения производительности. Когда мы столкнемся с необходимостью добавить неархитектурное состояние, мы обратим на это ваше внимание.

Архитектурное состояние — это информация, необходимая для определения текущего состояния компьютера и его действий. Если кто-то сохранит копию архитектурного состояния и содержимого памяти, выключит компьютер, затем снова включит его и восстановит архитектурное состояние и память, то компьютер возобновит выполнение программы, даже не подозревая, что его выключали. Это похоже на сюжет научно-фантастического романа, в котором мозг главного героя заморозили, а затем разморозили, и он проснулся в новом мире.

Чтобы микроархитектура оставалась простой для понимания, мы рассмотрим только небольшое подмножество набора команд RISC-V, а именно:

- арифметические и логические команды типа R: `add`, `sub`, `and`, `or`, `slt`;
- команды доступа в память: `lw`, `sw`;
- команды условного перехода: `beq`.

Это подмножество команд было выбрано потому, что его достаточно для разработки многих интересных программ. Как только вы поймете, как реализовать эти команды в аппаратуре, вы сможете добавить и другие.

7.1.2. Процесс разработки

Мы разделим нашу микроархитектуру на две взаимодействующие части: тракт данных и устройство управления. Тракт данных работает со словами данных. Он содержит такие блоки, как память, регистры, АЛУ и муль-

типлексоры. Мы реализуем 32-битную архитектуру RISC-V (RV32I), поэтому используем 32-битный тракт данных. Устройство управления получает текущую команду из тракта данных и в ответ говорит ему, как именно выполнять эту команду. В частности, устройство управления генерирует адресные сигналы для мультиплексоров, сигналы разрешения работы для регистров и сигналы разрешения записи в память.

Хороший способ разработки сложной системы – начать с элементов, которые хранят ее состояние. Эти элементы включают память для хранения команд и данных и блоки для хранения архитектурного состояния, т. е. счетчик команд и видимые программисту регистры[1]. Затем между этими элементами нужно расположить комбинационные схемы, вычисляющие новое состояние на основе текущего состояния. Команда читается из той части памяти, где находится программа; команды чтения из памяти и записи в память затем читают или записывают данные в другую часть памяти. Поэтому зачастую бывает удобно разделить память на две меньшие по размеру части, чтобы одна содержала команды, а другая – данные. На **рис. 7.1** показаны четыре вышеупомянутых элемента: счетчик команд, регистровый файл, память команд и память данных.

В этой главе самыми толстыми линиями обозначены 32-битные шины данных. Линии потоньше используются для шин меньшей разрядности, таких как пятибитная шина адреса регистрового файла. Самые тонкие синие линии применяются для управляющих сигналов, таких как сигнал разрешения записи в регистровый файл. Мы будем использовать линии разной толщины и дальше, чтобы избежать загромождения диаграмм указанием разрядности шин. Кстати, у элементов, хранящих состояние системы, обычно есть сигнал сброса, который устанавливает их в известное состояние в момент включения. Мы не будем показывать сигналы сброса на диаграммах.

Из всех регистров процессора как минимум счетчик команд (PC) должен иметь сигнал сброса, который проинициализирует его в момент включения процессора. При получении сигнала сброса процессор RISC-V инициализирует PC значением из области низких адресов памяти, например 0x00001000; как только сигнал сброса снят, процессор начинает выполнять код по этому адресу.

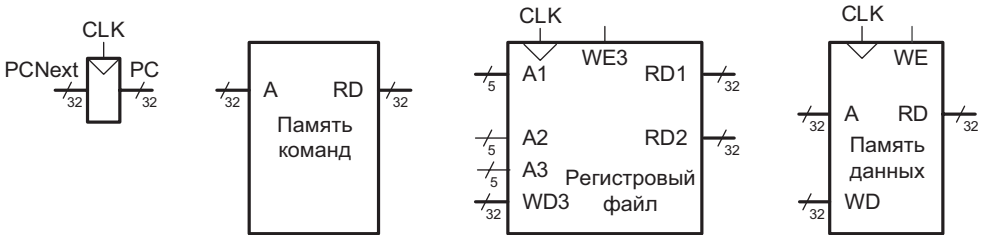


Рис. 7.1 Элементы, хранящие состояние процессора RISC-V

[1] В зависимости от контекста под состоянием процессора может пониматься как его чисто архитектурное состояние, так и архитектурное состояние плюс содержимое памяти. – *Прим. перев.*

Счетчик команд – это обычный 32-битный регистр, который указывает на текущую инструкцию. Его вход *PCNext* указывает адрес следующей инструкции, а выход *PC* содержит адрес текущей инструкции.

Память команд имеет единственный порт чтения[1]. На адресный вход A подается 32-битный адрес команды, после чего на выходе *RD* появляется 32-битное число, представляющее собой команду, прочитанное из памяти по этому адресу.

Регистровый файл содержит 32 элемента по 32 бита каждый – регистры `x0-x31`. Напомним, что регистр `x0` всегда содержит неизменяемое значение 0. Регистровый файл имеет два порта чтения и один порт записи данных. Порты чтения имеют пятибитные входы адреса *A1* и *A2*, каждый из которых определяет один из $2^5 = 32$ регистров в качестве источника данных для команды. Каждый из портов читает 32-битное значение из регистра и подает его на выходы *RD1* и *RD2* соответственно. Порт записи получает пятибитный адрес регистра на адресный вход *A3*, 32-битное число на вход данных *WD3*, сигнал разрешения записи *WE3* и тактовый сигнал. Если сигнал разрешения записи равен единице, то регистровый файл записывает данные в указанный регистр по положительному фронту тактового сигнала.

Память данных имеет единственный порт чтения/записи. Если сигнал разрешения записи *WE* равен единице, то данные с входа *WD* записываются в ячейку памяти с адресом *A* по положительному фронту тактового сигнала. Если же сигнал разрешения записи равен нулю, то данные из ячейки с адресом *A* подаются на выход *RD*.

Чтение из памяти команд, регистрового файла и памяти данных происходит асинхронно, то есть независимо от тактового сигнала. Другими словами, сразу же после изменения значения на адресном входе на выходе RD появляются новые данные. Это происходит не мгновенно, так как существует задержка распространения сигнала, при этом тактовый сигнал для чтения не требуется. Запись же производится исключительно по положительному фронту тактового сигнала. Таким образом, состояние системы изменяется только по фронту тактового сигнала. Адрес, данные и сигнал разрешения записи должны стать корректными за некоторое время до прихода фронта тактового сигнала (время предустановки, setup) и ни в коем случае не должны изменяться до тех пор, пока не пройдет некоторое время после прихода фронта (время удержания, hold).

В связи с тем, что элементы памяти изменяют свои значения только по положительному фронту тактового сигнала, они являются синхронными последовательностными схемами. Микропроцессор строится из

[1] Это упрощение сделано для того, чтобы можно было считать память команд памятью только для чтения (ROM); в большинстве реальных процессоров память команд должна быть доступна и для записи, чтобы операционная система могла загружать в нее новые программы. Многотактная микроархитектура, описанная в разделе 7.4, более реалистична в этом плане, так как содержит общую память команд и данных, доступную как для чтения, так и для записи.

тактируемых элементов памяти и комбинационной логики, поэтому он тоже является синхронной последовательностной схемой. На самом деле процессор можно рассматривать как гигантский конечный автомат или как несколько более простых и взаимодействующих между собой конечных автоматов.

7.1.3. Микроархитектуры RISC-V

В этой главе мы разработаем три микроархитектуры для процессорной архитектуры RISC-V: однотактную, многотактную и конвейерную. Они различаются тем, как связаны элементы состояния, а также наличием или отсутствием неархитектурного состояния.

Однотактная микроархитектура выполняет всю команду за один такт. Ее принцип работы легко объяснить, а устройство управления довольно простое. Из-за того, что все действия выполняются за один такт, эта микроархитектура не требует никакого неархитектурного состояния. Но длительность такта при этом ограничена самой медленной командой. Кроме того, процессору требуются отдельные запоминающие устройства для команд и данных, что на практике, как правило, избыточно.

К примерам классических многотактных процессоров можно отнести MIT Whirlwind 1947 года, IBM System/360, Digital Equipment Corporation VAX, 6502, используемый в Apple II, и 8088, используемый в IBM PC. Многотактные микроархитектуры по-прежнему используются в недорогих микроконтроллерах, например серии 8051, 68HC11 и PIC16, в бытовой технике, игрушках и гаджетах.

Многотактная микроархитектура выполняет команду за несколько более коротких тактов. Простым командам нужно меньше тактов, чем сложным. Вдобавок многотактная микроархитектура уменьшает количество необходимой аппаратуры путем повторного использования таких «дорогих» блоков, как сумматоры и блоки памяти. Например, при выполнении команды один и тот же сумматор на разных тактах может быть использован для разных целей. Повторное использование блоков достигается путем добавления в многотактный процессор нескольких неархитектурных регистров для записи в память промежуточных результатов. Многотактный процессор выполняет только одну команду за раз, и каждая команда занимает несколько тактов. Такой процессор обходится единственным запоминающим устройством, поскольку обращается к нему в одном цикле для получения команды, а в другом – для чтения или записи данных. Экономия на оборудовании стала определяющим фактором применения многотактных процессоров в недорогих системах.

Процессоры Intel стали конвейерными с момента выпуска 80486 в 1989 году. Почти все микропроцессоры RISC являются конвейерными, и к ним же относятся все коммерческие процессоры RISC-V. Из-за снижения стоимости транзисторов ядро конвейерного процессора теперь стоит доли цента, а вся система с памятью и периферийными устройствами стоит около 10 центов. Поэтому сегодня конвейерные процессоры заменяют своих более медленных многотактных собратьев даже в самых дорогостоящих приложениях.

Конвейерная микроархитектура – результат применения принципа конвейерной обработки к однотактной микроархитектуре. Вследствие этого она позволяет выполнять несколько команд одновременно,

значительно улучшая пропускную способность процессора. Конвейерная микроархитектура требует дополнительной логики для разрешения конфликтов между одновременно выполняемыми в конвейере командами. Она также требует несколько неархитектурных регистров, расположенных между стадиями конвейера. Тем не менее эта дополнительная логика и регистры того стоят – в наши дни все коммерческие высокопроизводительные процессоры используют конвейеры.

Мы изучим особенности и компромиссы этих трех микроархитектур в следующих разделах. В конце главы мы упомянем дополнительные способы увеличения производительности, используемые в современных высокопроизводительных процессорах.

Когда потребители выбирают себе компьютеры на основе бенчмарков, они должны быть осторожны, потому что производители компьютеров заинтересованы в завышении результатов. Например, бенчмарк Dhrystone содержит большое количество операций копирования строк, но эти строки имеют известную постоянную длину и выравнены по словам. Следовательно, продвинутый компилятор может заменить обычный код, состоящий из циклов и побайтовых операций чтения/записи, на последовательность чтения и записи слов, улучшая показатели Dhrystone более чем на 30 %, но не ускоряя работу реальных приложений. Бенчмарк SPEC89 содержал программу Matrix 300, в которой 99 % процессорного времени уходило на операции с одной строкой. IBM ускорила программу в 9 раз с помощью специально разработанного компилятора с так называемой технологией *блокировки*. Измерение производительности многоядерных систем является еще более сложным и неоднозначным процессом, потому что существует множество приемов разработки программ, позволяющих ускорить выполнение программы пропорционально количеству доступных ядер, но неэффективных на одном ядре. Другие программы работают быстро на одном ядре, но почти не выигрывают от дополнительных ядер.

7.2. Анализ производительности

Как мы уже упоминали ранее, у каждой процессорной архитектуры может быть много различных микроархитектур, обеспечивающих разное соотношение цены и производительности. Цена зависит от количества логических элементов и от технологии производства микросхемы. Прогресс в КМОП-технологиях позволяет размещать все больше и больше транзисторов на чипе за те же деньги, что активно используется для производства новых процессоров с еще большей производительностью. Точный расчет цены невозможен без детального знания конкретной технологии производства, но в целом чем больше логических элементов и памяти, тем выше цена.

В этом разделе мы познакомимся с основами анализа производительности. Производительность компьютерной системы можно измерить множеством способов, и маркетологи стараются выбрать именно те из них, которые позволяют их компьютерам выглядеть в наилучшем свете вне зависимости от того, имеют эти измерения какое-либо отношение к реальной жизни или нет. Например, производители микропроцессоров часто продают свою продукцию, акцентируя внимание потребителя на тактовой частоте и количестве ядер. Тем не менее в рекламе редко упоминают тот факт, что одни процессоры выполняют больше работы, чем другие с такой же тактовой частотой, и что этот эффект зависит от конкретной программы. Как же быть пользователю?

Единственный по-настоящему честный способ узнать производительность компьютера – измерить время выполнения вашей программы. Чем быстрее компьютер выполнит ее, тем выше его производительность. Еще один хороший способ – измерить время выполнения не одной, а нескольких программ, похожих на те, которые вы планируете запускать; это особенно важно, если ваша программа еще не разработана или измерения проводит кто-то, у кого ее нет. Такие программы называются *бенчмарками* (benchmark), а полученные результаты обычно публикуются, чтобы было ясно, насколько быстр компьютер.

Здесь стоит упомянуть три популярных бенчмарка – Dhrystone, CoreMark и SPEC. Первые два – это синтетические тесты, составленные из наиболее часто употребляемых фрагментов программ. Dhrystone был разработан в 1984 году и по-прежнему широко используется для встраиваемых процессоров, хотя его код несколько утратил актуальность для современных программ. CoreMark содержит более сложный и разнообразный набор кода по сравнению с Dhrystone, включая перемножение матриц, которым проверяют быстродействие умножителя и сумматора, связанные списки для проверки системы памяти, конечные автоматы для выполнения логики ветвления и циклические проверки избыточности, в которых задействованы многие блоки процессора. Оба бенчмарка имеют размер менее 16 КБ и не заполняют кеш команд.

Бенчмарк SPECspeed 2017 Integer от компании Standard Performance Evaluation Corporation (SPEC) состоит из реальных программ, включая x264 (сжатие видео), deepsjeng (игра в шахматы с искусственным интеллектом), omnetpp (моделирование) и GCC (компилятор C). Данный бенчмарк широко используется для высокопроизводительных процессоров, потому что он репрезентативно нагружает всю систему.

Время выполнения программы в секундах можно вычислить по формуле **(7.1)**:

$$\text{Время выполнения} = (\text{количество команд})\left(\frac{\text{такты}}{\text{команда}}\right)\left(\frac{\text{секунды}}{\text{такт}}\right). \quad (7.1)$$

Количество команд в программе зависит от архитектуры процессора. У некоторых архитектур могут быть очень сложные команды, каждая из которых выполняет множество действий, что уменьшает общее количество команд в программе. Но такие команды зачастую медленнее выполняются логическими схемами процессора. Количество команд также сильно зависит от квалификации программиста. В этой главе мы будем подразумевать, что количество команд в программах одинаково для всех реализаций архитектуры RISC-V, то есть не зависит от микроархитектуры. *Количество тактов на команду*, часто называемое CPI (cycles per instruction), – это среднее количество тактов процессора, необходимых для выполнения команды. Это соотношение обратно

пропорционально производительности, измеряемой в командах на такт (instructions per cycle, IPC). У разных микроархитектур разное CPI. В этой главе мы будем считать, что процессор работает с идеальной подсистемой памяти, которая никак не влияет на CPI. В **главе 8** мы рассмотрим случаи, когда процессору иногда приходится ждать ответа из памяти, что увеличивает CPI.

Количество секунд на такт – это период T_c тактового сигнала, который зависит от имеющей наибольшую задержку цепи, соединяющей логические элементы внутри процессора (критический путь). У разных микроархитектур период тактового сигнала может сильно отличаться. Он зависит в том числе и от выбранных разработчиками способов реализации аппаратных блоков. Например, сумматор с ускоренным переносом работает быстрее, чем сумматор с последовательным переносом. До сих пор улучшение технологий производства удваивало скорость переключения транзисторов каждые четыре–шесть лет, так что процессор, произведенный сегодня, работает гораздо быстрее, чем процессор с точно такой же микроархитектурой и аппаратными блоками, но произведенный десять лет назад.

Главная задача, стоящая перед разработчиком микроархитектуры, – создать такой процессор, который обеспечивал бы наименьшее возможное время выполнения программ, в то же время удовлетворяя ограничениям по цене и/или энергопотреблению. Так как решения, принятые на микроархитектурном уровне, влияют и на CPI, и на T_c, и в свою очередь зависят от выбранных аппаратных блоков и схемотехнических решений, то выбор наилучшего варианта требует очень внимательного анализа.

Существует много других факторов, которые влияют на общую производительность компьютера. Например, производительность жестких дисков, памяти, графической или сетевой подсистемы может быть настолько низкой, что производительность процессора на их фоне не будет иметь абсолютно никакого значения. Даже самый быстрый в мире процессор не поможет вам загружать веб-сайты быстро, если вы подключены к интернету через обычную телефонную линию. Эти факторы выходят за рамки данной книги, и рассматривать их мы не будем.

7.3. Однотактный процессор

Сначала мы разработаем микроархитектуру, которая выполняет команды за один такт. Начнем с конструирования тракта данных путем соединения приведенных на **рис. 7.1** элементов, хранящих состояние процессора, при помощи комбинационной логики, которая и будет выполнять разные команды. Управляющие сигналы нужны, чтобы указывать, как именно тракт данных должен выполнять команду, находящуюся в нем в текущий момент времени. Устройство управления содержит комбинационную логику, которая формирует необходимые управляющие сигна-

лы в зависимости от того, какая команда выполняется в данный момент. В заключение мы оценим производительность такого процессора.

7.3.1. Пример программы

Для большей конкретики наш однотактный процессор будет выполнять короткую программу (**рис. 7.2**), которая выполняет чтение из памяти, запись в память, инструкцию типа *R* (`or`) и условный переход (`beq`). Предположим, что программа хранится в памяти начиная с адреса 0x1000. На рисунке указан адрес каждой команды, а также ее тип, поля и шестнадцатеричный машинный код.

Предположим, что регистр `x5` изначально содержит значение 6, а регистр `x9` – значение 0x2004. Ячейка памяти 0x2000 содержит значение 10. Счетчик программ начинается с 0x1000. Команда `lw` читает значение 10 из адреса памяти (0x2004 – 4) = 0x2000 и помещает его в регистр `x6`. Команда `sw` записывает 10 по адресу (0x2004 + 8) = 0x200C. Команда `or` вычисляет значение в регистре `x4` = 6 | 10 = 01102 | 10102 = 11102 = 14. Затем команда условного перехода `beq` возвращается к метке `L7`, поэтому программа выполняется бесконечно.

7.3.2. Однотактный тракт данных

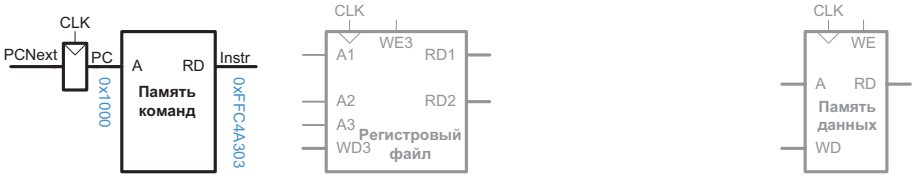
В этом разделе мы шаг за шагом создадим однотактный тракт данных, используя элементы, показанные на **рис. 7.1**. Новые элементы и цепи будем выделять черным (или синим, в случае управляющих сигналов), а уже рассмотренные элементы будем перекрашивать серым. Пример выполняемой команды показан внизу каждого рисунка.

Счетчик команд (program counter, PC) содержит адрес команды, которую надо выполнить. На первом этапе нам надо прочитать эту команду из памяти команд. Как показано на **рис. 7.3**, счетчик команд напрямую подключен к адресному входу памяти команд. Команда, прочитанная, или выбранная (fetched), из памяти команд, – это 32-битная команда, отмеченная на рисунке как *Instr*. В нашем примере программы на **рис. 7.2** PC = 0x1000. (Обратите внимание, что у нас 32-разрядный процессор, поэтому на самом деле PC = 0x00001000, но мы опускаем ведущие нули, чтобы облегчить чтение.)

Мы выделяем курсивом названия *сигналов*, но не названия аппаратных модулей. Например, *PC* – это сигнал, выходящий из регистра PC, или просто счетчик команд (PC).

Адрес	Команда	Тип	Поля						Машинный код
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Рис. 7.2 Пример программы, выполняющей различные типы команд



Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

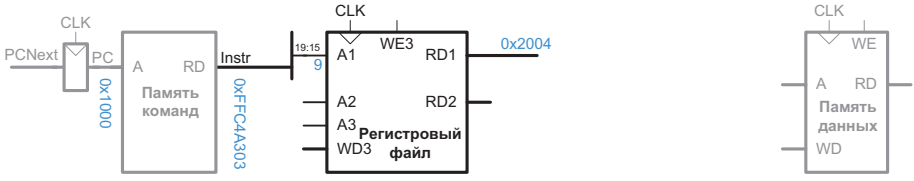
Рис. 7.3 Выборка команды из памяти

В нашем примере программы Instr – это `lw` (чтение слова из памяти), или на машинном языке 0xFFC4A303, как показано в нижней части **рис. 7.3**. Эти значения, применяемые в качестве примера, помечены на рисунке голубым цветом.

Дальнейшие действия процессора будут зависеть от того, какая именно команда была выбрана. Для начала давайте создадим тракт данных для команды `lw`, после чего подумаем, как расширить его так, чтобы он мог выполнять и другие команды.

Команда `lw`

Для команды `lw` на следующем этапе мы должны прочитать регистр операнда (source register), содержащий так называемый базовый адрес. Номер этого регистра указан в поле `rs1` ($Instr_{19:15}$). Эти пять бит подключены к адресному входу первого порта ($A1$) регистрового файла, как показано на **рис. 7.4**. Значение, прочитанное из регистрового файла, появляется на его выходе *RD1*. В нашем примере регистровый файл читает значение 0x2004 из регистра `x9`.

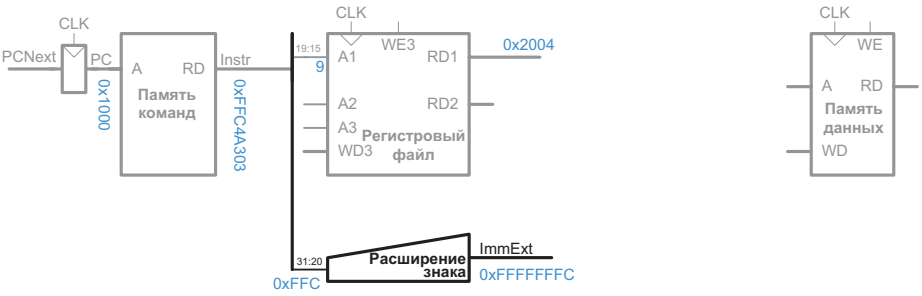


Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Рис. 7.4 Чтение операнда из регистрового файла

Команде `lw` также требуется смещение (offset) – число, которое будет прибавлено к базовому адресу. Смещение передается как непосредственный операнд в 12-битном поле $Instr_{31:20}$. Так как это число может быть как положительным, так и отрицательным, то над ним должна быть выполнена операция знакового расширения до 32 бит. Знаковое расши-

рение заключается в том, что знаковый бит (он же старший бит) расширяемого числа просто копируется во все старшие биты расширенного числа, а именно $ImmExt_{31:12} = Instr_{31}$ и $ImmExt_{11:0} = Instr_{31:20}$. Расширение знака выполняется специальным модулем, который получает 12-битное число со знаком в $Instr_{31:20}$ и выдает 32-битную расширенную знаком константу *ImmExt*, как показано на **рис. 7.5**. В нашем примере представленная в дополнительном коде константа −4 расширяется из ее 12-битного представления 0xFFC до 32-битного представления 0xFFFFFFFC.



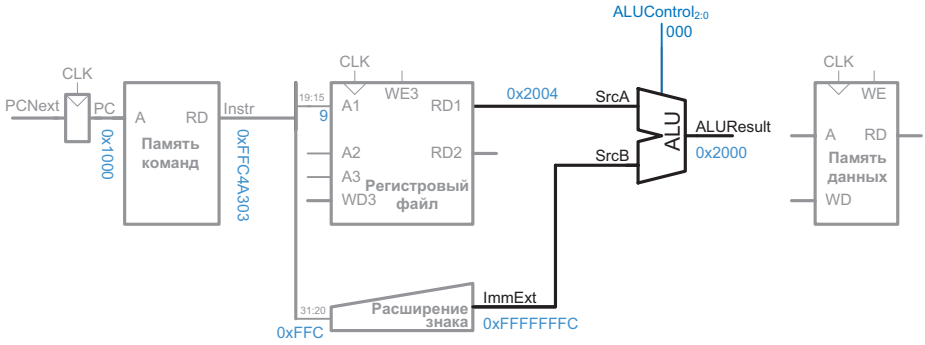
Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Рис. 7.5 Знаковое расширение непосредственного операнда

Процессор должен добавить смещение к базовому адресу, чтобы получить адрес, по которому будет произведено чтение из памяти. Для выполнения операции сложения мы добавляем в тракт данных АЛУ (ALU), как показано на **рис. 7.6**. АЛУ получает на входы два операнда, *SrcA* и *SrcB*. Операнд *SrcA* – это базовый адрес из регистрового файла, а *SrcB* – это смещение со знаковым расширением *ImmExt*. АЛУ может выполнять множество операций, о которых говорилось в **разделе 5.2.4**. Трехбитный управляющий сигнал *ALUControl* говорит АЛУ, какую операцию надо выполнить (**табл. 5.3**). АЛУ получает 32-битные операнды и генерирует 32-битный результат *ALUResult*. Для команды `lw` сигнал *ALUControl* должен быть равен 000 – в этом случае смещение будет прибавлено к базовому адресу. *ALUResult* отправляется в память данных как адрес для чтения, как показано на **рис. 7.6**. В нашем примере АЛУ выполняет вычисление 0x2004 + 0xFFFFFFFC = 0x2000. Это тоже 32-битное значение, но мы опускаем ведущие нули, чтобы не усложнять чтение.

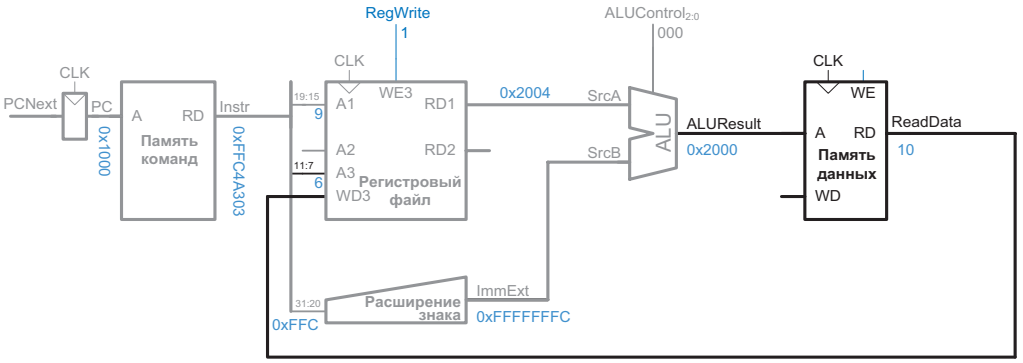
Далее *ALUResult* подается на адресный вход памяти данных (*A*). Значение, прочитанное из памяти данных, попадает на шину *ReadData*, после чего записывается обратно в регистровый файл в конце такта, как показано на **рис. 7.7**. Третий порт регистрового файла – это порт записи. Регистр результата `lw`, обозначенный как поле `rd` ($Instr_{11:7}$), подклю-

чен к адресному входу третьего порта (*A*3) регистрового файла. Шина *ReadData* подключена к входу данных третьего порта (*WD*3). Управляющий сигнал *RegWrite* (запись в регистр), в свою очередь, соединен с входом разрешения записи третьего порта (*WE*3) и активен во время выполнения команды `lw`, чтобы прочитанное значение было записано в регистровый файл. Сама запись происходит по положительному фронту тактового сигнала, которым заканчивается такт процессора. В нашем примере процессор извлекает значение 10 из адреса 0x2000 в памяти данных и помещает это значение в `x6` в регистровом файле.



Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Рис. 7.6 Вычисление адреса данных в памяти

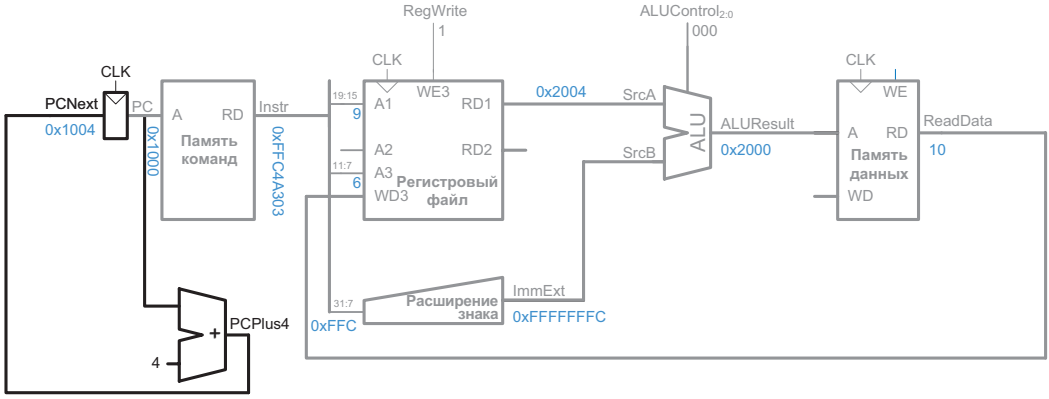


Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Рис. 7.7 Чтение памяти и запись результата обратно в регистровый файл

Одновременно с выполнением команды процессор должен вычислить адрес следующей команды, *PCNext*. Так как команды 32-битные (четы-

рехбайтные), то адрес следующей команды равен *PC* + 4. На **рис. 7.8** показан еще один сумматор для увеличения *PC* на 4. В нашем примере *PCNext* = 0x1000 + 4 = 0x1004. Новый адрес записывается в программный счетчик по следующему переднему фронту тактового сигнала. На этом создание тракта данных для инструкции `lw` завершено.



Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

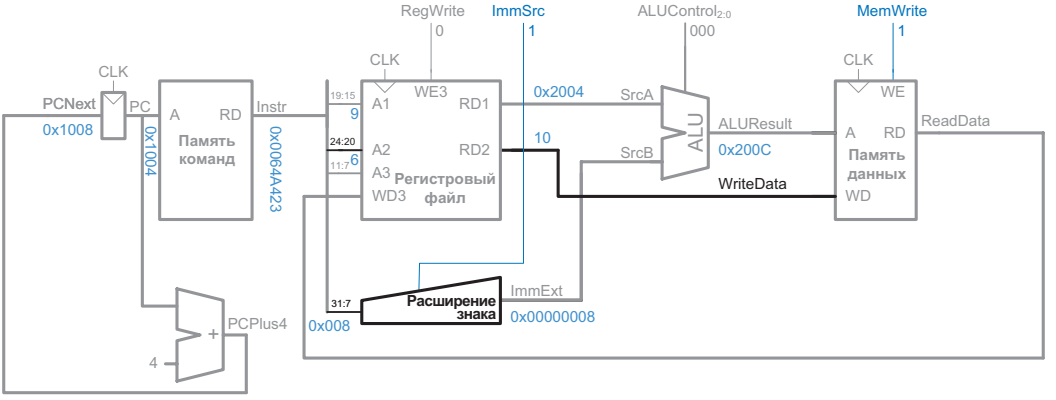
Рис. 7.8 Увеличение счетчика команд

Команда `sw`

Теперь давайте доработаем тракт данных, чтобы он мог выполнять еще и команду `sw`. Как и `lw`, команда `sw` читает базовый адрес из первого порта регистрового файла и расширяет знаковым битом смещение, передаваемое как непосредственный операнд. АЛУ складывает базовый адрес со смещением, чтобы получить адрес в памяти. Все эти функции уже реализованы в тракте данных, но 12-битная константа со знаком хранится в битах $Instr_{31:25,11:7}$ вместо $Instr_{31:20}$, как это было для `lw`. Следовательно, модуль расширения знака нужно модифицировать таким образом, чтобы он принимал дополнительные биты $Instr_{11:7}$. Для простоты (и с учетом будущих команд, таких как `jal`) модуль расширения принимает все биты $Instr_{31:7}$. Управляющий сигнал *ImmSrc* определяет, какие биты команды будут использованы в качестве битов константы. Когда *ImmSrc* = 0 (команда `lw`), модуль расширения рассматривает $Instr_{31:20}$ в качестве 12-битной константы со знаком; когда *ImmSrc* = 1 (команда `sw`), модуль выбирает биты $Instr_{31:25,11:7}$.

Команда `sw`, в отличие от `lw`, читает из регистрового файла еще один регистр и записывает его содержимое в память данных. На **рис. 7.9** показаны дополнительные соединения для этой новой функции. Номер регистра указывается в поле `rs2` ($Instr_{24:20}$), которое подключено ко второму порту (*A2*) регистрового файла. Прочитанное значение поступает на

второй выход (*RD2*) и попадает на вход записи в память данных. Вход разрешения записи (*WE*) управляется сигналом *MemWrite*. Для команды `sw` сигнал *MemWrite* = 1, чтобы данные были записаны в память; *ALUControl* = 000, чтобы базовый адрес был просуммирован со смещением; и *RegWrite* = 0, потому что команда ничего не пишет в регистровый файл. Заметьте, что *ReadData* читается из памяти в любом случае, но прочитанное значение игнорируется, так как *RegWrite* = 0.



Адрес	Команда	Тип	Поля					Машинный код
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

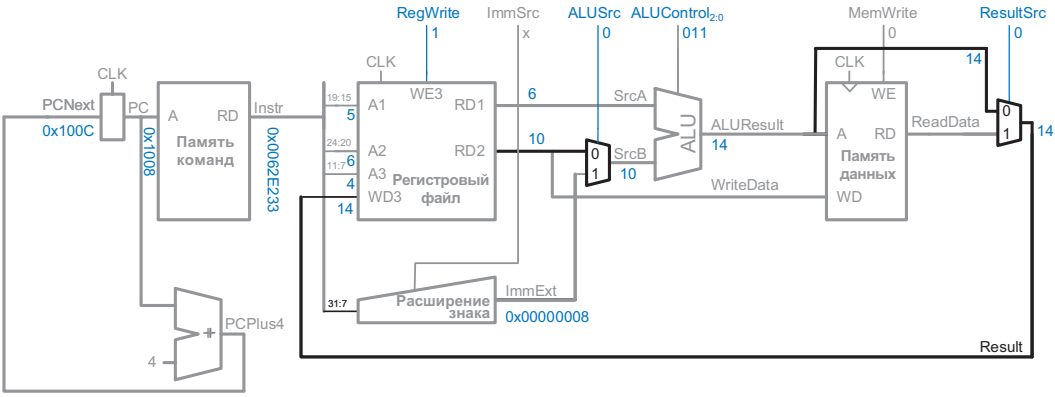
Рис. 7.9 Запись командой `sw` данных в память

В нашем примере счетчик команд PC = 0x1004. Следовательно, из памяти команд будет прочитана команда `sw` с машинным кодом 0x0064A423. Регистровый файл извлекает значение 0x2004 (базовый адрес) из регистра `x9` и значение 10 из `x6`, в то время как модуль расширения увеличивает разрядность непосредственного смещения 8 с 12 до 32 бит. АЛУ выполняет вычисление 0x2004 + 8 = 0x200C. Модуль памяти данных записывает значение 10 по адресу 0x200C. В то же время значение PC увеличивается до 0x1008.

Команды типа *R*

Теперь добавим поддержку команд типа R – `add`, `sub`, `and`, `or` и `slt`. Все эти команды читают два регистра из регистрового файла, выполняют над ними некие операции в АЛУ и записывают результат обратно в третий регистр. Единственное различие этих команд – в типе операции. Таким образом, все они могут быть выполнены одной и той же аппаратурой, используя лишь разные значения управляющего сигнала *ALUControl*. Напомним, что в **разделе 5.2.4** приведены значения *ALUControl* – 000 для сложения, 001 для вычитания, 010 для AND, 011 для OR и 101 для команды «установить, если меньше» (`slt`).

На **рис. 7.10** показан тракт данных, доработанный для выполнения инструкций типа *R*. Значения *RS*1 и *RS*2 считываются из первого и второго портов регистрового файла и поступают на входы АЛУ. Нам пришлось добавить в схему мультиплексор и новый управляющий сигнал *ALUSrc*, который позволяет выбирать между *ImmExt* и *RD*2 в качестве второго источника АЛУ, *SrcB*. Для команд `lw` и `sw` *ALUSrc* = 1 и вторым операндом является дополненная константа *ImmExt*; для инструкций типа *R* *ALUSrc* = 0 и на вход *SrcB* подается значение из регистрового файла *RD*2.



Адрес	Команда	Тип	Поля						Машинный код
			funct7	rs2	rs1	f3	rd	op	
0x1008	`or x4, x5, x6`	R	0000000	00110	00101	110	00100	0110011	0062E233

Рис. 7.10 **Изменения в тракте данных для поддержки команд типа *R***

Значение, которое будет записано обратно в регистровый файл, мы будем называть *Result*. После выполнения команды `lw` результат получается из вывода *ReadData* памяти. Но для инструкций R-типа *Result* поступает из вывода *ALUResult* блока ALU. Мы добавляем мультиплексор результатов, чтобы выбрать правильный результат в зависимости от типа инструкции. Сигнал выбора мультиплексора *ResultSrc* равен 0 для инструкций R-типа, чтобы выбрать *ALUResult* в качестве результата; *ResultSrc* равен 1 для `lw`, чтобы выбрать *ReadData*. Нас не волнует значение *ResultSrc* для `sw`, потому что эта инструкция ничего не записывает в регистровый файл.

На **рис. 7.9** порт записи регистрового файла был всегда подключен к памяти данных. Но команды типа *R* должны записывать в регистровый файл значение *ALUResult*. Чтобы выбирать между *ReadData* и *ALUResult*, мы добавили еще один мультиплексор, выход которого назвали *Result*. Этот мультиплексор управляется еще одним новым сигналом – *ResultSrc*. Сигнал *ResultSrc* равен нулю для команд типа *R* – в этом случае *Result* принимает значение *ALUResult*. Для команды `lw` сигнал

Обратите внимание, что наша реализация процессора вычисляет все возможные результаты, полученные в ходе выполнения команд (например, *ALUResult* и *ReadData*), а затем использует мультиплексор, чтобы выбрать нужный результат на основе кода команды. Это важная стратегия разработки процессоров. В оставшейся части данной главы мы продолжим добавлять мультиплексоры для выбора результата. Одно из основных различий между программным и аппаратным обеспечением заключается в том, что программное обеспечение работает последовательно, поэтому мы можем вычислить именно тот ответ, который нам нужен. Цифровая схема работает параллельно; поэтому мы часто вычисляем все возможные ответы, а затем выбираем тот, который нам нужен. Например, при выполнении инструкции типа *R* с помощью АЛУ модуль памяти все равно получает адрес и извлекает данные по этому адресу, даже если они не нужны.

ResultSrc равен единице, а *Result* принимает значение *ReadData*. Для команды `sw` значение *ResultSrc* не играет никакой роли, так как `sw` ничего не записывает в регистровый файл.

В нашем примере PC = 0x1008. Поэтому из памяти команд читается команда `or` с машинным кодом 0x0062E233. Регистровый файл считывает исходные операнды 6 из регистра `x5` и 10 из регистра `x6`. *ALUControl* = 011, поэтому АЛУ выполняет вычисление 6 | 10 = 01102 | 10102 = 11102 = 14. Результат записывается в регистр `x4`. В то же время значение PC увеличивается до 0x100C.

Команда `beq`

Добавим поддержку команды `beq`. Эта команда сравнивает два регистра, и если они равны, то добавляет смещение к счетчику команд PC, выполняя, таким образом, условный переход.

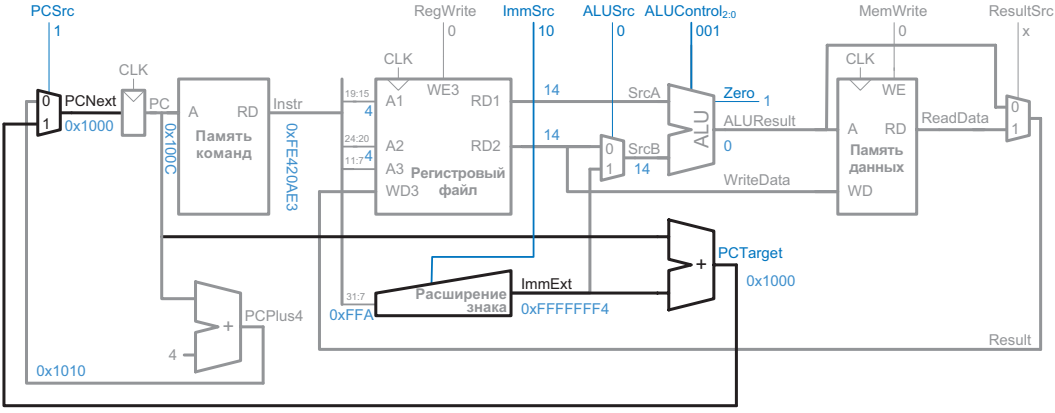
Смещение условного перехода представляет собой 13-битную положительную или отрицательную константу, которая хранится в 12-битном поле константы команды типа *B*. Следовательно, логике расширения нужен еще один режим, чтобы выбрать правильные биты константы. Сигнал *ImmSrc* увеличивается до 2 бит и работает в соответствии с кодировкой из **табл. 7.1**. Результат *ImmExt* теперь получается либо путем расширения константы знаком (когда *ImmSrc* = 00 или 01), либо прибавлением смещения условного перехода (когда *ImmSrc* = 10).

Таблица 7.1 Кодировка ImmSrc

ImmSrc	ImmExt	Тип	Описание
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-битная константа со знаком
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-битная константа со знаком
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-битная константа со знаком

На **рис. 7.11** показаны изменения в канале данных. Нам нужен еще один сумматор для вычисления целевого адреса перехода *PCTarget* = *PC* + *ImmExt*. Два регистра сравниваются путем вычитания *SrcA* – *SrcB* в АЛУ. Если *ALUResult* равен 0, на что указывает флаг нуля, то регистры равны. Кроме того, нужно добавить мультиплексор, чтобы выбрать, какое именно значение присвоить *PCNext* – *PCPlus4* или *PCTarget*. Значение *PCTarget* используется, если выполняется команда условного перехода и установлен флаг нуля. Для команды `beq` управляющий сигнал *ALUControl* = 001, поэтому АЛУ выполняет вычитание. *ALUSrc* =

0, чтобы операнд *SrcB* был прочитан из регистрового файла. Сигналы *RegWrite* и *MemWrite* равны 0, потому что команда условного перехода ничего не записывает ни в регистровый файл, ни в память. Значение *ResultSrc* нас не интересует, поскольку запись в регистровый файл не происходит.



Адрес	Команда	Тип	Поля					Машинный код
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	`beq x4, x4, L7`	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Рис. 7.11 Изменения в тракте данных для поддержки команды `beq`

В нашем примере *PC* = 0x100C, поэтому из памяти извлекается команда `beq` с машинным кодом 0xFE420AE3. Оба исходных операнда извлекаются из регистра `x4`, поэтому регистровый файл читает 14 на обоих портах. АЛУ выполняет вычисление 14 − 14 = 0 и устанавливает флаг нуля. Тем временем модуль расширения выдает значение 0xFFFFFFF4 (т. е. −12), которое складывается с содержимым PC, чтобы получить *PCTarget* = 0x1000. Обратите внимание, что старшие 12 бит 13-битного кода константы показаны на входе модуля расширения (0xFFA). Мультиплексор PCNext выбирает *PCTarget* в качестве следующего значения PC и переходит обратно к началу кода по следующему положительному фронту тактового сигнала.

На этом разработка тракта данных однотактного процессора завершена. Мы рассмотрели не только устройство процессора, но и сам процесс разработки, во время которого выбирали элементы памяти и соединяли их при помощи все усложняющейся комбинационной логики. В следующем разделе мы рассмотрим, как формировать управляющие сигналы, настраивающие тракт данных на выполнение той или иной команды.

Теоретически мы могли бы построить модуль расширения знаком из 32-битного мультиплексора 3:1, выбрав один из трех возможных входов на основе *ImmSrc* и различных битовых полей инструкции. На практике старшие биты расширенной знаком константы всегда получаются из бита 31 (поле команды $Instr_{31}$), поэтому мы можем оптимизировать схему и использовать мультиплексор только для выбора младших битов.

Мы называем мультиплексоры по имени сигнала, который они производят на выходе. Например, мультиплексор PCNext генерирует сигнал *PCNext*, а мультиплексор Result генерирует сигнал *Result*.

7.3.3. Однотактный блок управления

Блок управления формирует управляющие сигналы на основе полей `op`, `funct3` и `funct7`. В наборе команд RV32I используется только бит 5 функции `funct7`, поэтому мы будем оперировать битами `op` ($Instr_{6:0}$), `funct3` ($Instr_{14:12}$) и $\texttt{funct7}_5$ ($Instr_{30}$). На **рис. 7.12** показан однотактный процессор с блоком управления, подключенным к тракту данных.

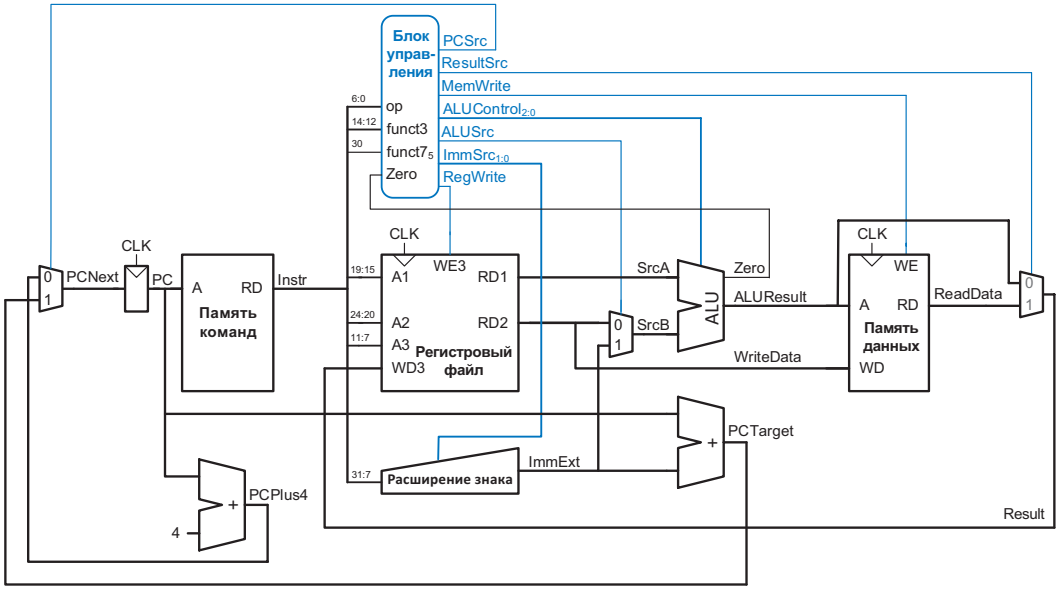


Рис. 7.12 Однотактный процессор

На **рис. 7.13** показана иерархическая структура *блока управления*, который также называют *контроллером*, или *дешифратором*, поскольку он расшифровывает машинный код команды. Блок управления можно условно разделить на две основные части: основной дешифратор, который вырабатывает большую часть управляющих сигналов, и дешифратор АЛУ, который решает, какую операцию будет выполнять АЛУ.

В **табл. 7.2** показаны управляющие сигналы, которые производит основной дешифратор в соответствии с разработанным нами ранее трактом данных. Основной дешифратор определяет тип инструкции по коду команды, а затем генерирует соответствующие управляющие сигналы для тракта данных. Основной дешифратор генерирует большинство управляющих сигналов для тракта данных, а также внутренние сигналы *Branch* и *ALUOp* для собственных нужд блока управления. Схему основного дешифратора можно разработать на основе таблицы истинности, используя приемы разработки комбинационной логики, которыми вы уже владеете.

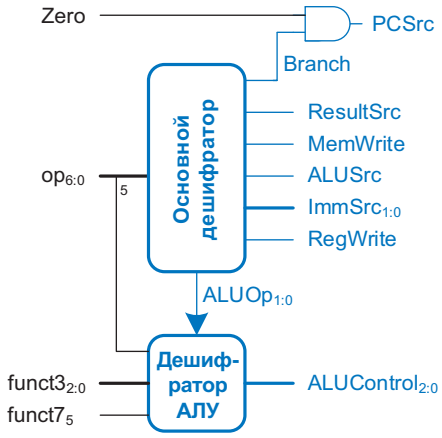


Рис. 7.13 **Внутренняя структура блока управления однотактным процессором**

Таблица 7.2 **Таблица истинности основного дешифратора**

Команда	op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
тип *R*	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01

Дешифратор АЛУ вырабатывает управляющий сигнал *ALUControl* на основе внутреннего сигнала *ALUOp* и внешних данных `funct3`. Если встречаются команды `sub` и `add`, то для выработки сигнала *ALUControl* дешифратор АЛУ также использует биты $\texttt{funct7}_5$ и $\texttt{op}_5$ в соответствии с **табл. 7.3**.

Таблица 7.3 **Таблица истинности дешифратора АЛУ**

ALUOp	funct3	{op_5, $funct7_5$}	ALUControl	Команда
00	x	x	000 (сложение)	`lw, sw`
01	x	x	001 (вычитание)	`beq`
10	000	00, 01, 10	000 (сложение)	`add`
	000	11	001 (вычитание)	`sub`
	010	x	101 (установить, если меньше)	`slt`
	110	x	011 (ИЛИ)	or
	111	x	010 (И)	and

Сигнал *ALUOp* = 00 указывает на сложение (например, чтобы найти адрес для чтения из памяти или записи в память). ALUOp = 01 указывает на вычитание (например, для сравнения двух чисел при условном переходе). ALUOp = 10, указывает на команду АЛУ типа *R*, в которой де-

Согласно табл. B.1 из приложения B, команды `add`, `sub` и `addi` содержат биты `funct3` = 000. Команда `add` также содержит биты `funct7` = 0000000, а `sub` содержит биты `funct7` = 0100000, поэтому бита $\texttt{funct7}_5$ достаточно, чтобы различать эти две команды. Но вскоре мы рассмотрим схему выполнения команды `addi`, у которой нет поля `funct7`, но зато есть поле `op` = 0010011. Мы увидим, что команда АЛУ, у которой `funct3` = 000, — это `sub`, если $\texttt{op}_5 = \texttt{funct7}_5 = 1$, или в противном случае одна из команд `add` или `addi`.

шифратор АЛУ должен проанализировать поле `funct3` (а иногда также биты $\texttt{op}_5$ и $\texttt{funct7}_5$), чтобы решить, какую операцию должно выполнить АЛУ (например, `add`, `sub`, `and`, `or`, `slt`).

Пример 7.1 ФУНКЦИОНИРОВАНИЕ ОДНОТАКТНОГО ПРОЦЕССОРА

Определите значения управляющих сигналов и частей канала данных, которые используются при выполнении инструкции `and`.

Решение На рис. 7.14 показаны управляющие сигналы и пути движения данных во время выполнения команды `and`. Счетчик команд указывает на ячейку памяти, из которой выбирается команда; модуль памяти команд извлекает и выводит эту команду. Прохождение данных через регистровый файл и АЛУ показано синей толстой линией. Из регистрового файла читаются два исходных операнда, определяемых сигналом *Instr*. Операнд *SrcB* должен поступать из второго порта регистрового файла (не *ImmExt*), поэтому сигнал *ALUSrc* должен быть равен нулю. АЛУ выполняет побитовую операцию И, поэтому *ALUControl* должен быть равен 010. Сигнал *Result* формируется в АЛУ, поэтому *ResultSrc* должен быть равен нулю, а результат записывается в регистровый файл, поэтому *RegWrite* должен быть равен единице. Команда ничего не пишет в память, так что сигнал *MemWrite* должен быть равен нулю.

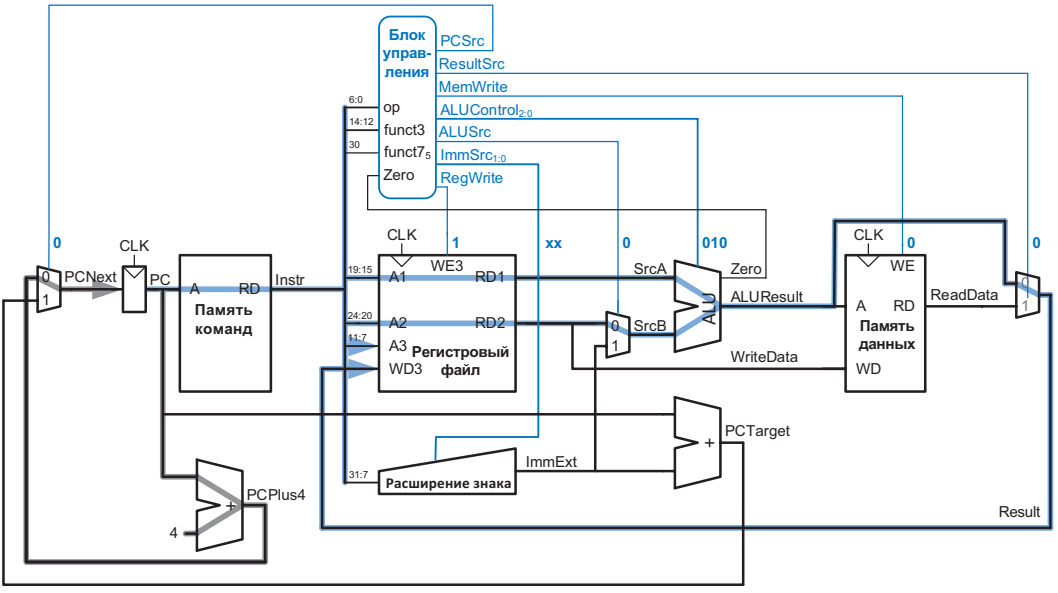


Рис. 7.14 Управляющие сигналы и пути движения данных при выполнении команды `and`

Так как команда `and` не является командой условного перехода, то сигнал *Branch* равен нулю и, соответственно, *PCSrc* тоже равен нулю. В результате счетчик команд получает новое значение из *PCPlus4*. Путь, по которому происходит обновление значения PC с помощью сигнала *PCPlus4*, показан толстой серой линией. Важно учитывать, что по цепям, которые не выделены на рисунке, тоже передаются какие-то сигналы и данные, но для этой конкретной команды совершенно не имеет значения, что они из себя представляют. Например, происходит расширение знака непосредственного операнда, а данные читаются из памяти, но это не оказывает никакого влияния на будущее состояние системы.

7.3.4. Дополнительные команды

Мы рассмотрели лишь небольшое подмножество полной системы команд RISC-V. В этом разделе мы доработаем тракт данных и блок управления для поддержки инструкций `addi` (сложение с непосредственным операндом) и `jal` (безусловный переход с возвратом). Тем самым мы сформируем систему команд, достаточную для разработки множества интересных программ. Приложив немало усилий, мы могли бы расширить однотактный процессор для обработки каждой инструкции RISC-V. Мы увидим, что поддержка некоторых новых команд зачастую заключается всего лишь в усложнении основного дешифратора, тогда как для других команд могут понадобиться дополнительные аппаратные блоки в тракте данных.

Пример 7.2 КОМАНДА `addi`

Напомним, что `addi rd, rs1, imm` – это команда типа *I*, которая складывает значение в `rs1` с расширенной знаком константой и записывает результат в `rd`. В тракте данных уже есть вся необходимая функциональность для выполнения этой команды. Определите, какие изменения необходимо внести в устройство управления, чтобы добавить поддержку команды `addi`.

Решение Все, что нужно сделать, – это добавить новую строку в таблицу истинности основного дешифратора и заполнить ее значениями управляющих сигналов для команды `addi`, как показано в **табл. 7.4**. Так как результат должен быть записан в регистровый файл, то *RegWrite* должен быть равен единице. 12-битное значение константы в $Instr_{31:20}$ расширено знаком, как это уже было в случае `lw`, другой команды типа *I*, поэтому *ImmSrc* = 00 (**табл. 7.1**). На вход *SrcB* подается непосредственный операнд, поэтому *ALUSrc* = 1. Так как команда `addi` не является командой условного перехода, а также не пишет в память, то сигналы *MemWrite* = *Branch* = 0. Результат формируется в АЛУ, а не читается из памяти, так что *ResultSrc* = 0. Наконец, АЛУ должно выполнить сложение, поэтому сигнал *ALUOp* = 10; дешифратор АЛУ генерирует сигнал *ALUControl* = 000, потому что `funct3` = 000 и $op_5 = 0$.

Таблица 7.4 Таблица истинности основного дешифратора с поддержкой `addi`

Команда	op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
тип *R*	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01
`addi`	**0010011**	1	00	1	0	0	0	10

Проницательный читатель может заметить, что благодаря внесенным изменениям у нас появилась возможность выполнять другие команды типа *I*: `andi`, `ori` и `slti`. Все эти команды имеют одно и то же значение `op` = 0010011, нуждаются в одних и тех же управляющих сигналах и отличаются только полем `funct3`, которое дешифратор АЛУ уже использует для генерации сигнала *ALUControl* и, таким образом, определяет операцию АЛУ.

Пример 7.3 Команда `jal`

Покажите, как изменить однотактный процессор RISC-V для поддержки команды перехода с возвратом `jal`, которая записывает *PC* + 4 в `rd` и заменяет значение в PC на целевой адрес перехода, *PC* + `imm`.

Решение Процессор вычисляет целевой адрес перехода, значение *PCNext*, выполняя сложение *PC* и 21-битной константы со знаком, непосредственно закодированной в команде. Наименьший значащий бит константы всегда равен 0, а следующие 20 наиболее старших значащих битов берутся из $Instr_{31:12}$. Затем эта 21-битная константа расширяется знаковым битом. В тракте данных уже есть все необходимое для сложения *PC* и расширенной знаком константы, записи полученного значения в *PC*, вычисления *PC* + 4 и записи результата в регистровый файл. Следовательно, в тракте данных осталось лишь модифицировать модуль расширения знака, чтобы он мог обработать 21-битное значение константы и расширить мультиплексор сигнала *Result*, дабы тот мог выбрать *PC* + 4 (т. е. *PCPlus4*), как показано на **рис. 7.15**. В **табл. 7.5** представлена новая кодировка сигнала *ImmSrc* с поддержкой длинной константы, необходимой для команды `jal`.

Таблица 7.5 Кодировка ImmSrc

ImmSrc	ImmExt	Тип	Описание
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-битная константа со знаком
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-битная константа со знаком
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-битная константа со знаком
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-битная константа со знаком

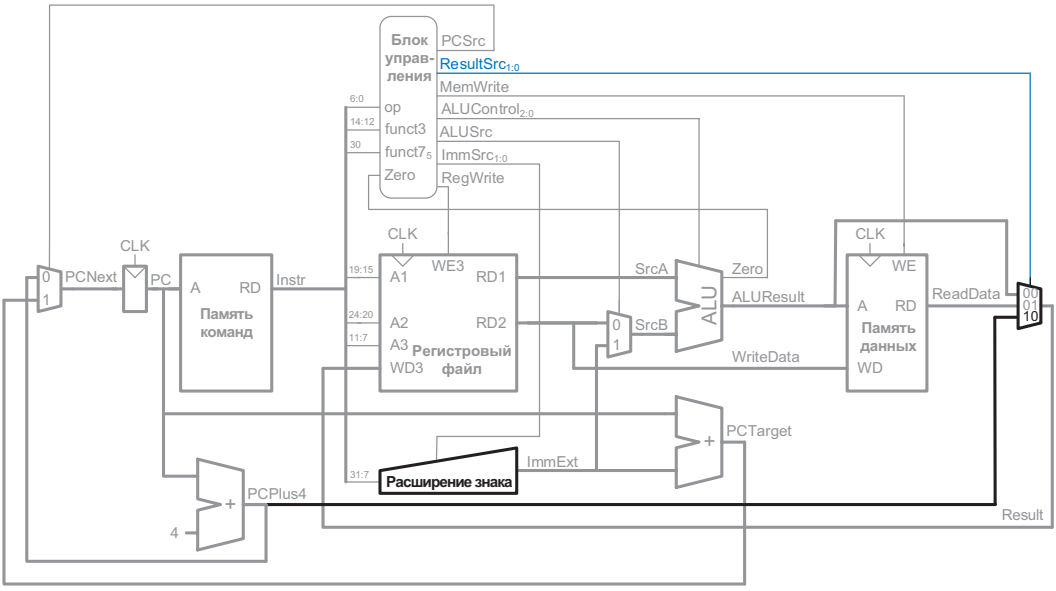


Рис. 7.15 **Изменения в тракте данных для поддержки команды `jal`**

Для команды безусловного перехода блок управления должен установить сигнал *PCSrc* = 1. Для этого мы добавляем в схему логический элемент ИЛИ и новый управляющий сигнал *Jump*, как показано на **рис. 7.16**. Когда сигнал *Jump* принимает значение логической единицы, то *PCSrc* также становится равен единице и следующее значение *PC* загружается из *PCTarget* (целевой адрес перехода).

В **табл. 7.6** показана обновленная таблица истинности основного дешифратора, в которую мы добавили новую строку для команды `jal` и новый столбец для сигнала *Jump*. Для записи *PC* + 4 в регистр `rd` сигнал *RegWrite* = 1, а сигнал *ResultSrc* = 10. Сигнал *ImmSrc* = 11 и указывает на 21-битное смещение перехода. Сигналы *ALUSrc* и *ALUOp* могут быть любыми, потому что нас не интересует результат вычислений в АЛУ. Сигнал *MemWrite* = 0, потому что инструкция не пишет в память,

Таблица 7.6 **Таблица истинности основного дешифратора с поддержкой команды `jal`**

Команда	op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	1	0	00	0
`sw`	0100011	0	01	1	1	x	0	00	0
тип *R*	0110011	1	xx	0	0	0	0	10	0
`beq`	1100011	0	10	0	0	x	1	01	0
тип *I*	0010011	1	00	1	0	00	0	10	0
`jal`	**1101111**	1	11	x	0	10	0	xx	1

а *Branch* = 0, потому что команда не является переходом по условию. Новый сигнал *Jump* равен единице и тем самым указывает, что следующим значением *PC* должен быть целевой адрес перехода.

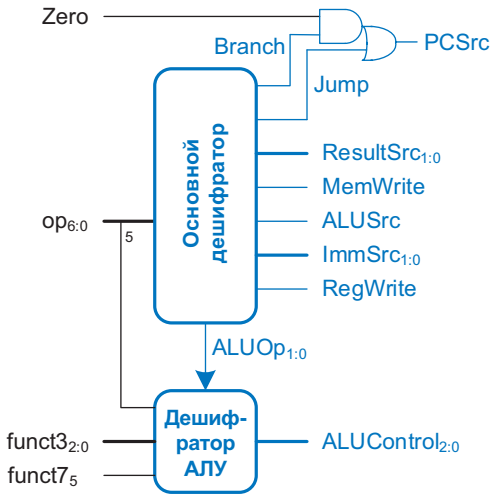


Рис. 7.16 Расширенный блок управления для команды `jal`

7.3.5. Анализ производительности

Вспомним **уравнение 7.1**, из которого следует, что время выполнения программы вычисляется как произведение количества команд, количества тактов на команду и длительности такта. Каждая инструкция в однотактном процессоре выполняется ровно за один такт, поэтому количество тактов на команду (cycles per instruction, CPI) составляет 1. Минимальная длительность такта определяется *цепью с наибольшей задержкой* (критическим путем). В нашем процессоре команда `lw` выполняется дольше всех и использует критический путь, показанный на **рис. 7.17** толстыми синими линиями. Он начинается там, где в счетчик команд по положительному фронту тактового сигнала записывается новое значение. Блок памяти команд извлекает новую команду, а регистровый файл подает значение `rs1` на вход *SrcA*. Во время чтения регистрового файла поле константы расширяется знаком в соответствии с управляющим сигналом *ImmSrc* и через мультиплексор поступает на вход *SrcB* (этот путь выделен серым цветом). АЛУ складывает *SrcA* и *SrcB*, чтобы найти адрес памяти. Блок памяти данных читает содержимое ячейки памяти по этому адресу, а мультиплексор результата передает *ReadData* в *Result*. Наконец, сигнал *Result* должен успеть установиться на входе регистрового файла до того, как придет следующий положительный фронт тактового сигнала, иначе будет записано неверное значение. Таким образом, минимальная длительность одного такта составляет:

$$T_{c\_single} = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + tRF_{setup}. \quad (7.2)$$

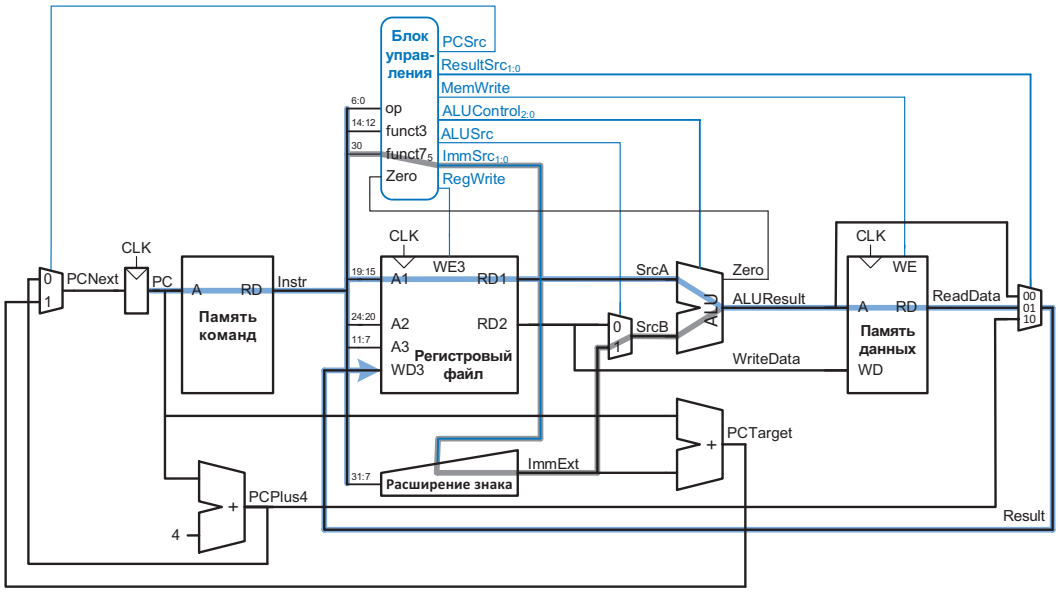


Рис. 7.17 Критический путь для `lw`

Напомним, что команда `lw` не использует второй порт чтения (A2/RD2) регистрового файла.

В большинстве технологий производства микросхем доступ к АЛУ, памяти и регистровым файлам занимает гораздо больше времени, чем прочие операции. Следовательно, на самом деле критический путь проходит через регистровый файл, а не через дешифратор, модуль расширения и мультиплексор. Этот путь выделен толстой синей линией на **рис. 7.17**. Таким образом, мы можем приближенно посчитать длительность одного такта как

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}. \quad (7.3)$$

Численное значение длительности такта зависит от конкретной технологии.

У других команд цепи с наибольшей задержкой могут быть короче. Например, командам типа *R* не нужно обращаться к памяти данных. Тем не менее раз уж мы разрабатываем синхронные последовательностные схемы, то период тактового сигнала всегда должен определяться самой медленной командой.

Пример 7.4 ПРОИЗВОДИТЕЛЬНОСТЬ ОДНОТАКТНОГО ПРОЦЕССОРА

Бен Битдидл задумал построить однотактный процессор по 7-нм КМОП-техпроцессу. Он выяснил, что задержки логических элементов такие же, как в **табл. 7.7**. Помогите ему вычислить время выполнения программы, состоящей из 100 млрд команд.

Решение Согласно уравнению **(7.3)**, длительность такта однотактного процессора равна:

$$T_{c\_single} = 40 + 2(200) + 100 + 120 + 30 + 60 = 750 \text{ пс}.$$

Согласно уравнению **(7.1)** общее время выполнения программы составит:

$$T_{single} = (100 \times 10^9 \text{ команд})(1 \text{ такт} / \text{команда})(750 \times 10^{-12} \text{ с} / \text{такт}) = 75 \text{ с}.$$

Таблица 7.7 Задержки элементов

Элемент	Параметр	Задержка (пс)
Задержка распространения сигналов от входов к выходам (clk-to-Q) в регистре	t_{pcq}	40
Время предустановки регистра	t_{setup}	50
Мультиплексор	t_{mux}	30
Элемент И-ИЛИ	$t_{AND\text{-}OR}$	20
АЛУ	t_{ALU}	120
Дешифратор (блок управления)	t_{dec}	25
Блок расширения	t_{ext}	35
Чтение из памяти	t_{mem}	200
Чтение из регистрового файла	t_{RFread}	100
Время предустановки регистрового файла	$t_{RFsetup}$	60

7.4. Многотактный процессор

У однотактного процессора есть три заметных недостатка. Во-первых, ему требуется отдельная память для команд и данных, что зачастую нереально. В большинстве компьютеров используют общую память для команд и данных, доступную для чтения и записи. Во-вторых, период его тактового сигнала должен быть достаточно большим, чтобы успела выполниться самая медленная команда (lw), несмотря на то что большинство остальных команд гораздо быстрее. Наконец, ему нужно три сумматора (один для АЛУ и два для вычисления нового значения счетчика команд); сумматоры, особенно быстрые, требуют множества логических элементов, что делает их относительно дорогими схемами.

Один из способов решить эти проблемы – использовать многотактный процессор, в котором выполнение каждой команды происходит в не-

сколько этапов. Память, АЛУ и регистровый файл являются источниками самых больших задержек, поэтому для записи в память примерно одинаковой задержки на каждом коротком этапе процессор может задействовать только один из этих модулей. Процессор сможет обходиться общей памятью для команд и данных. Команды будут выбираться на первом этапе, а чтение или запись данных будут происходить на одном из последующих этапов. Кроме того, процессору понадобится только один сумматор; на разных этапах он может использоваться для разных целей. У разных команд в этом случае будет разное количество этапов, так что простые команды смогут выполняться быстрее, чем сложные.

Мы будем разрабатывать многотактный процессор тем же способом, что и однотактный. Сначала сконструируем тракт данных, соединяя при помощи комбинационной логики блоки памяти и блоки, хранящие архитектурное состояние процессора. Помимо этого, мы добавим и другие блоки для хранения информации о промежуточном (неархитектурном) состоянии между этапами. После этого займемся устройством управления. Так как теперь оно должно формировать разные управляющие сигналы в зависимости от текущего этапа выполнения команды, то вместо комбинационных схем нам понадобится конечный автомат. Напоследок мы снова оценим производительность и сравним ее с производительностью однотактного процессора.

7.4.1. Многотактный тракт данных

Как и прежде, в основу нашей разработки мы положим приведенные на **рис. 7.18** элементы, хранящие состояние – память и архитектурное состояние процессора. В однотактном процессоре мы использовали раздельную память для команд и данных, потому что нужно было за один и тот же такт читать из памяти команд и обращаться к памяти данных. Теперь мы будем использовать общую память, хранящую и команды, и данные. Это более реалистичный сценарий, и сейчас он возможен благодаря тому, что мы можем выбирать команду на одном такте, а обращаться к памяти данных на другом. Счетчик команд и регистровый файл при этом не изменились.

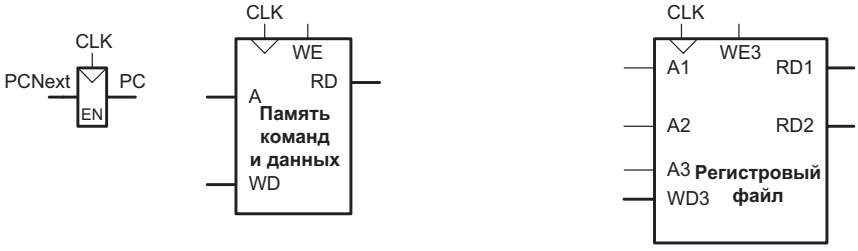


Рис. 7.18 Общая память команд и данных и элементы схемы, хранящие архитектурное состояние

Как и в однотактном процессоре, мы называем мультиплексоры и неархитектурные регистры тем же именем, что и сигналы, которые они генерируют. Например, регистр инструкций выдает сигнал *Instr*, а мультиплексор Result выдает сигнал *Result*.

Как и в случае с однотактным процессором, шаг за шагом мы будем добавлять новые компоненты, нужные для каждого из этапов выполнения команды. Счетчик команд содержит адрес команды, которая должна быть выполнена следующей. Соответственно, первым делом надо прочитать ее из памяти команд. Как показано на **рис. 7.19**, счетчик команд напрямую подсоединен к адресному входу памяти команд. Прочитанная из памяти команда сохраняется во временный (неархитектурный) регистр команд (Instruction Register, IR), так что мы сможем использовать ее в следующих тактах. Сигнал разрешения записи в регистр команд назовем *IRWrite* и будем использовать его, когда потребуется обновить находящуюся в регистре команду.

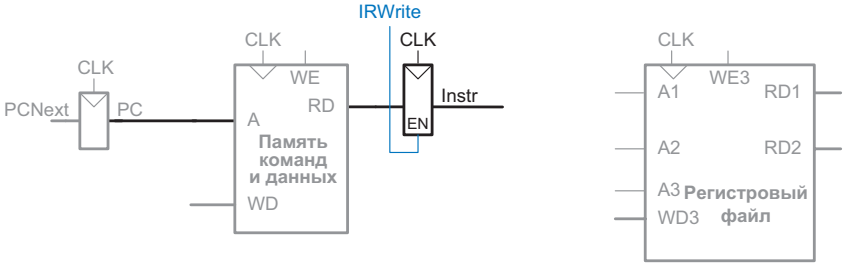


Рис. 7.19 Выборка команды из памяти

Команда `lw`

Как и в случае с однотактным процессором, мы сначала разработаем тракт данных для команды `lw`. После выборки команды `lw` вторым этапом является чтение регистра-источника, содержащего базовый адрес. Номер регистра указывается в поле `rs1` ($Instr_{19:15}$) и подается на адресный вход первого порта (*A1*) регистрового файла, как показано на **рис. 7.20**. Значение, прочитанное из регистрового файла, появляется на его выходе RD1, после чего сохраняется в другой неархитектурный регистр A.

Для инструкции `lw` также требуется 12-битное смещение расположенной в поле константы $Instr_{31:20}$, которое должно быть расширено знаком до 32 бит, как показано на **рис. 7.20**. Как и в однотактном процессоре, блок расширения знаком получает 2-битный управляющий сигнал *ImmSrc*, указывающий на 12-, 13- или 21-битный непосредственный операнд для различных типов команд. 32-битный расширенный непосредственный операнд называется *ImmExt*. Мы могли бы сохранить *ImmExt* в еще один временный (неархитектурный) регистр, но так как *ImmExt* – это выход комбинационной схемы, вход которой зависит исключительно от *Instr*, а это значение не будет меняться все то время, пока команда выполняется, то нет смысла добавлять еще один временный регистр для хранения константы.

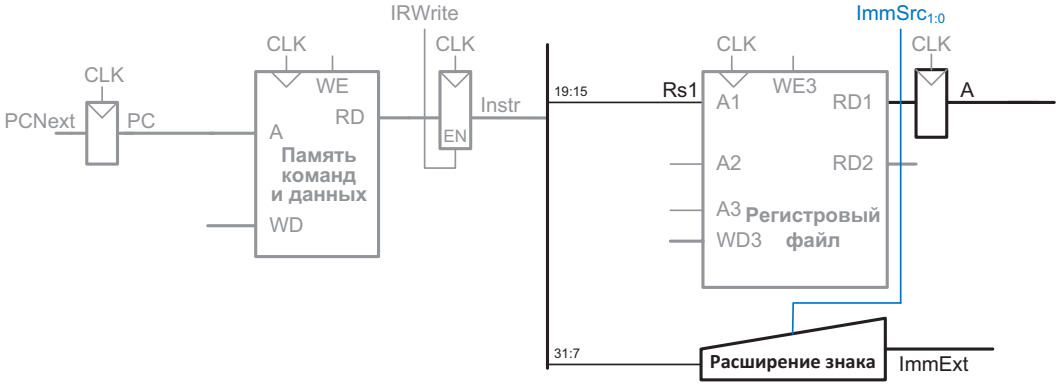


Рис. 7.20 Считывание одного операнда из регистрового файла и знаковое расширение второго операнда из поля константы

Адрес, по которому мы должны читать из памяти, получается путем сложения базового адреса и смещения. Для сложения мы используем АЛУ, как показано на **рис. 7.21**. Чтобы АЛУ выполнило сложение, управляющий сигнал *ALUControl* должен быть равен 000. *ALUResult* сохраняется во временном регистре *ALUOut*.

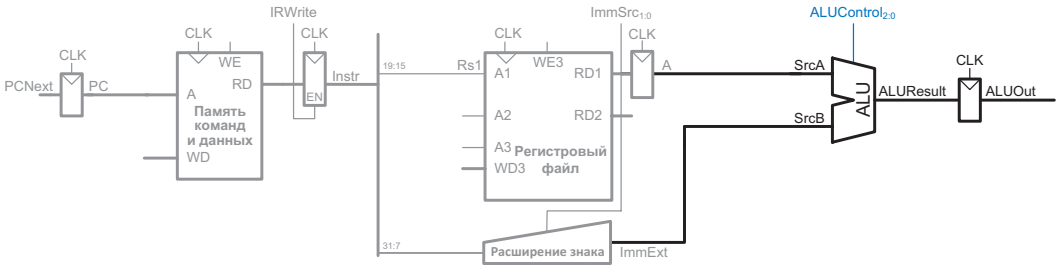


Рис. 7.21 Сложение базового адреса и смещения

На четвертом этапе мы должны прочитать данные из памяти, используя только что вычисленный адрес. Для этого перед адресным входом памяти необходимо добавить мультиплексор, чтобы в качестве адреса *Adr* можно было использовать либо *PC*, либо *ALUOut*, как показано на **рис. 7.22**. Прочитанные из памяти данные сохраняются во временном регистре *Data*. Заметьте, что мультиплексор адреса (Adr) позволяет нам повторно использовать память во время выполнения команды `lw`. На первом этапе в качестве адреса мы используем *PC*, что позволяет выбрать команду. На четвертом этапе в качестве адреса мы используем *ALUOut* и читаем данные. Следовательно, управляющий сигнал *AdrSrc* должен принимать разные значения на разных этапах выполнения команды. В **разделе 7.4.2** мы создадим конечный автомат, который будет формировать требуемую последовательность управляющих сигналов.

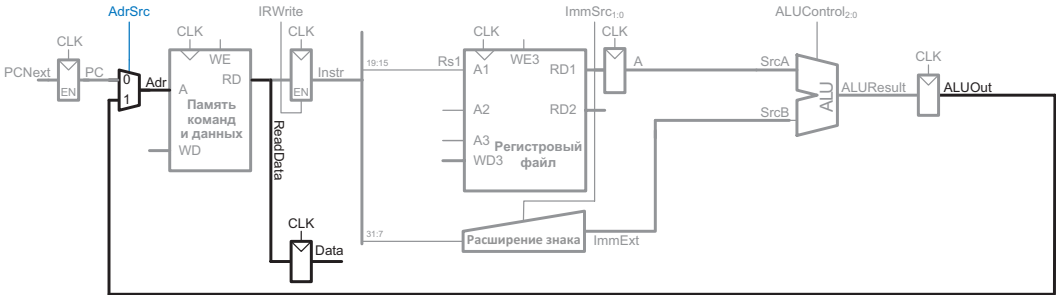


Рис. 7.22 Загрузка данных из памяти

На последнем этапе данные должны быть записаны в регистровый файл, как показано на **рис. 7.23**. Номер регистра результата определяется полем `rd` ($Instr_{11:7}$). Результат поступает из регистра Data. Вместо того чтобы подключать регистр данных напрямую ко входу записи *WD3* регистрового файла, мы добавим мультиплексор на шину *Result*, чтобы иметь возможность выбрать либо *ALUOut*, либо *Data*, перед тем как подать *Result* на вход записи регистрового файла (*WD*3). Нам это пригодится в будущем, потому что другим командам потребуется записать в регистровый файл результат из АЛУ. Сигнал *RegWrite* = 1 говорит о том, что регистровый файл должен быть обновлен.

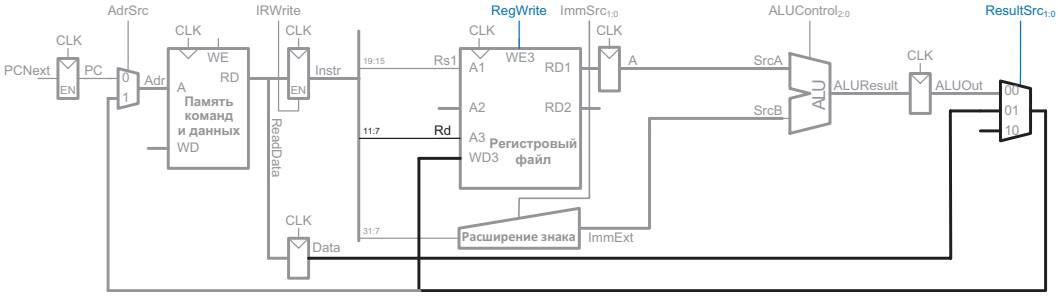


Рис. 7.23 Запись данных в регистровый файл

За то время, пока выполняются все вышеперечисленные операции, процессор должен увеличить значение счетчика команд на четыре. В однотактном процессоре для этого нам потребовался отдельный сумматор. В многотактном процессоре мы можем использовать уже имеющееся АЛУ на одном из первых этапов, пока оно еще не используется. Для этого понадобится добавить пару мультиплексоров, которые позволят подавать на входы АЛУ содержимое счетчика команд *PC* и константу 4, как показано на **рис. 7.24**. Мультиплексор, управляемый сигналом *ALUSrcA*, подает на вход *SrcA* либо *PC*, либо регистр *A*. Другой мультиплексор подает на вход *SrcB* либо константу 4, либо *ImmExt*. Оставшиеся входы

мультиплексора нам понадобятся позже, когда мы будем добавлять новые команды. Для обновления PC блок АЛУ добавляет *SrcA* (PC) к *SrcB* (4), и результат записывается в счетчик программ. Мультиплексор результатов выбирает эту сумму из ALUResult, а не из ALUOut; для этого требуется третий вход мультиплексора. Для того чтобы обновить счетчик команд, АЛУ складывает *SrcA* (*PC*) и *SrcB* (4) и записывает полученный результат в счетчик команд. Управляющий сигнал *PCWrite* разрешает запись в счетчик команд только на тех тактах, где это необходимо. На этом создание тракта данных для команды `lw` завершено.

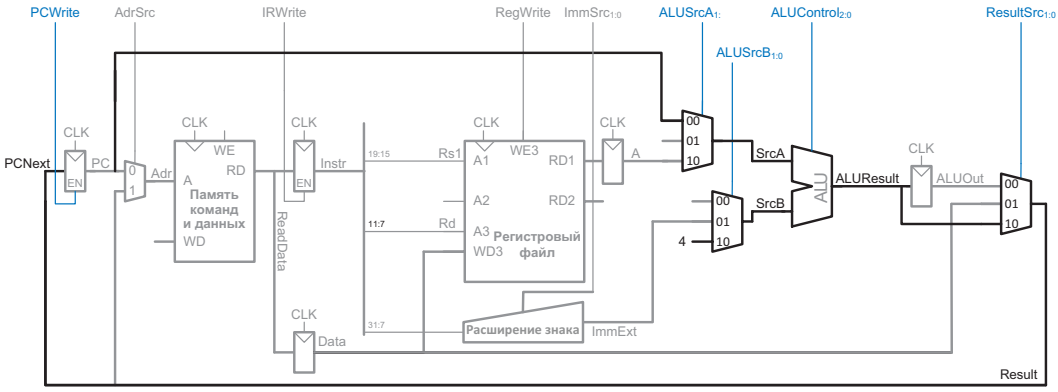


Рис. 7.24 **Увеличение счетчика команд на четыре**

Команда `sw`

Теперь дополним тракт данных для обработки команды `sw`. Как и команда `lw`, `sw` читает базовый адрес из первого порта регистрового файла и выполняет знаковое расширение непосредственного операнда, после чего АЛУ складывает их на втором этапе, получая адрес для записи в память на третьем этапе. Единственное отличие `sw` – это то, что мы должны прочитать еще один регистр из регистрового файла и записать его содержимое в память, как показано на **рис. 7.25**. Номер регистра указан в поле `rs2` ($Instr_{24:20}$), которое подключено ко второму порту (*A2*) регистрового файла. Прочитанное значение сохраняется во временном регистре *WriteData*, а из него подается на порт записи данных в память (*WD*) для записи на четвертом этапе. Новый управляющий сигнал *MemWrite* показывает, когда именно данные должны быть записаны в память.

Команды типа *R*

Команды типа *R* читают из регистрового файла два операнда и записывают результат обратно в регистровый файл. Тракт данных уже содержит все соединения, необходимые для этих этапов.

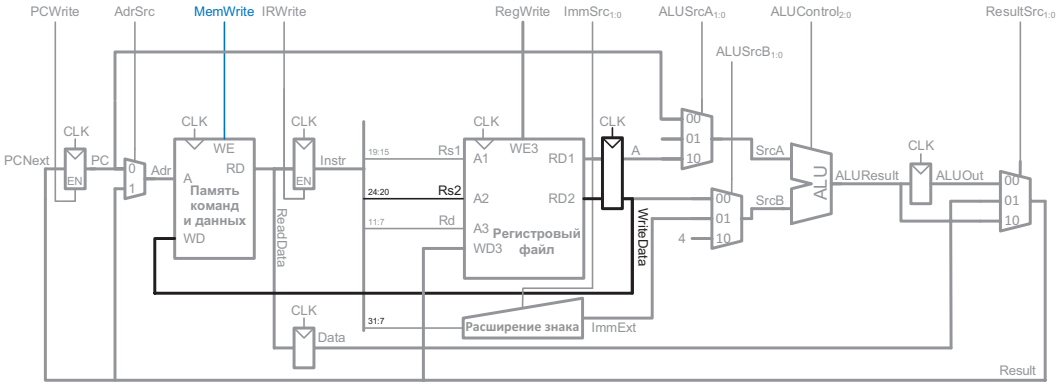


Рис. 7.25 Изменения в тракте данных для поддержки команды `sw`

Команда beq

Команда `beq` проверяет, равны ли два операнда, и вычисляет новое значение счетчика команд, складывая текущее значение *PC* с 13-битным смещением со знаком. Необходимые компоненты для сравнения регистров путем вычитания уже имеются в тракте данных.

На втором этапе выполнения команды нам не требуется АЛУ, поэтому мы используем его для вычисления целевого адреса условного перехода $PCTarget = PC + ImmExt$. К этому моменту команда извлечена из памяти, и значение *PC* уже обновлено до $PC + 4$. Следовательно, на первом этапе выполнения команды старое значение счетчика команд *OldPC* необходимо сохранить во временном регистре. На втором этапе АЛУ вычисляет $PC + ImmExt$, используя имеющиеся значения *OldPC* на входе *SrcA* и *ImmExt* на входе *SrcB*. В этот момент управляющий сигнал $ALUControl = 000$, что означает операцию сложения. Процессор сохраняет эту сумму в регистре *ALUOut*. На **рис. 7.26** показан обновленный тракт данных для поддержки команды `beq`.

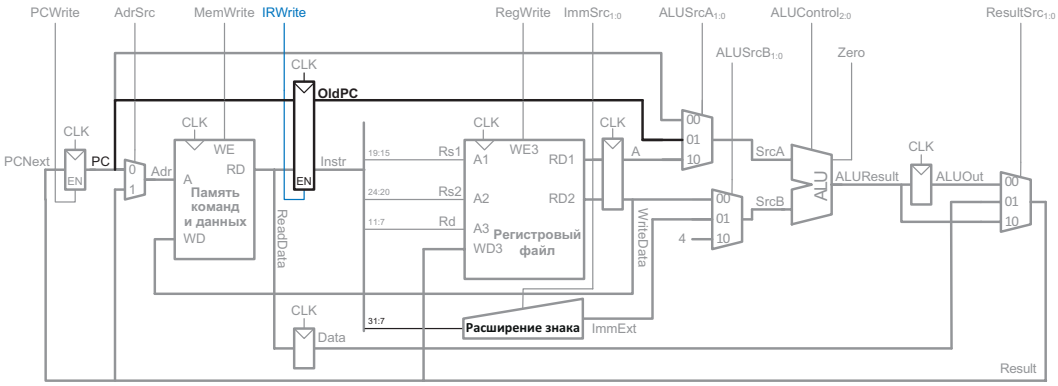


Рис. 7.26 Изменения в тракте данных для поддержки команды `beq`

На третьем этапе АЛУ вычитает один операнд из другого и устанавливает флаг нуля, если они равны. Если это так, то блок управления устанавливает в единицу сигнал разрешения записи в счетчик команд *PCWrite*, а мультиплексор результатов выбирает временный регистр *ALUOut* в качестве источника адреса перехода, и отправляет его значение в *PC*. Никакого нового оборудования не требуется.

На этом разработка многотактного тракта данных завершена. Процесс разработки был очень похож на тот, который мы использовали для однотактного процессора, когда постепенно добавляли блок за блоком между элементами, хранящими состояние процессора. Главное же отличие заключается в том, что каждая команда выполняется в несколько этапов. Нам потребовались не видимые программисту временные (неархитектурные) регистры, чтобы сохранять результаты каждого из этих этапов. За счет этого мы смогли повторно использовать одно и то же АЛУ, что позволило избавиться от нескольких сумматоров. Таким же образом мы смогли поместить команды и данные в общую память. В следующем разделе мы создадим конечный автомат, который будет формировать управляющие сигналы для каждого этапа в нужной последовательности.

7.4.2. Многотактное устройство управления

Как и в однотактном процессоре, устройство управления формирует управляющие сигналы в зависимости от полей `op`, `funct3` и `funct75` инструкции ($Instr_{6:0}$, $Instr_{14:12}$ и $Instr_{30}$ соответственно). На **рис. 7.27** показан многотактный процессор с устройством управления, подключенным к тракту данных. Тракт данных показан черным цветом, а устройство управления – синим.

Как показано на **рис. 7.28**, блок управления состоит из главного конечного автомата, дешифратора АЛУ и дешифратора команд. Дешифратор АЛУ такой же, как и в однотактном процессоре (**табл. 7.3**), но вместо комбинационного основного дешифратора однотактного процессора нам понадобится основной конечный автомат для генерации последовательности управляющих сигналов при поэтапном выполнении команды. Небольшой дешифратор команд комбинационно вырабатывает сигнал выбора непосредственного операнда *ImmSrc* на основе кода операции в соответствии со столбцом *ImmSrc* в **табл. 7.6.** Мы разрабатываем главный автомат как машину Мура, так что выходы являются только функцией текущего состояния. В оставшейся части этого раздела мы займемся разработкой диаграммы переходов между состояниями для главного конечного автомата.

Главный конечный автомат формирует сигналы управления мультиплексорами и сигналы разрешения записи в регистры тракта данных. Чтобы сделать диаграмму состояний более удобочитаемой, мы будем

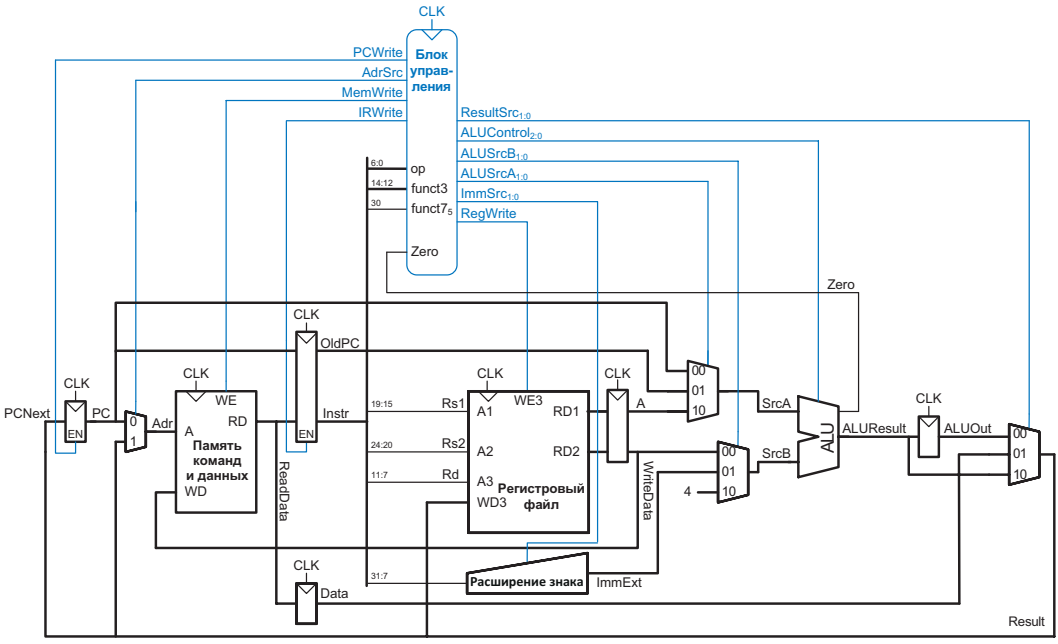


Рис. 7.27 Многотактный процессор

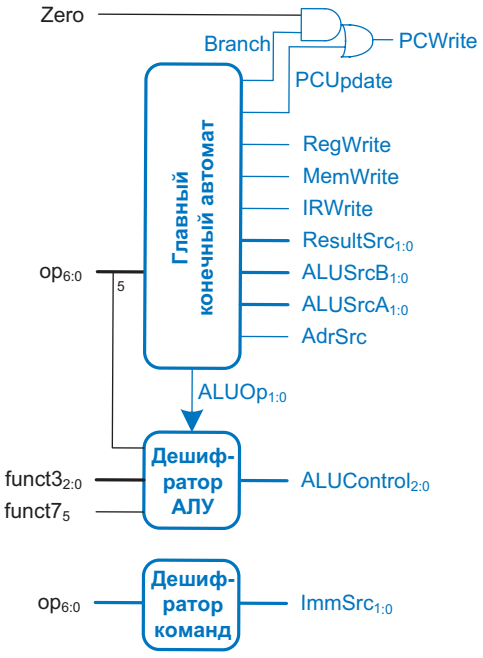


Рис. 7.28 Блок управления многотактным процессором

указывать только те управляющие сигналы, которые имеют смысл на конкретном этапе выполнения команды. Сигналы управления мультиплексорами будем указывать лишь тогда, когда они действительно используются. Разрешающие сигналы (*RegWrite*, *MemWrite*, *IRWrite*, *PCUpdate* и *Branch*) отображаются только тогда, когда они равны единице; в противном случае они равны нулю.

Выборка команды

Первым этапом выполнения любой команды является чтение из памяти по адресу, находящемуся в счетчике команд, то есть выборка команды из памяти (Fetch). В это состояние управляющий автомат переходит по сигналу сброса (*Reset*). Управляющие сигналы показаны на **рис. 7.29**. Для чтения команды из памяти *AdrSrc* = 0, поэтому адрес берется из PC. Чтобы прочитанное значение попало в регистр команд (IR), *IRWrite* устанавливается в единицу. Одновременно с этим текущее значение PC записывается в регистр OldPC. Путь данных через тракт для этого и следующих двух этапов команды `lw` показан на **рис. 7.32**, при этом поток данных во время этапа выборки выделен серым цветом.

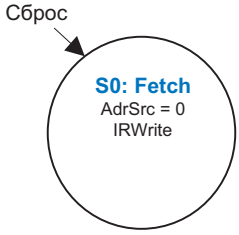


Рис. 7.29 Выборка команды

Декодирование команды

Второй этап – чтение регистрового файла и декодирование команды (Decode). Процессор определяет, какая операция должна быть выполнена на основе полей `op`, `funct3` и $\texttt{funct7}_5$. На этом этапе процессор также читает регистры-источники `rs1` и `rs2` и помещает считанные значения во временные регистры *A* и *WriteData*. Для декодирования управляющие сигналы не нужны. На **рис. 7.30** показано состояние главного конечного автомата при декодировании, а на **рис. 7.32** синими линиями средней толщины показан путь данных через тракт в этом состоянии. Завершив выполнение этапа декодирования команды, процессор может определить свои дальнейшие действия, поскольку команда была прочитана и декодирована. Сначала мы покажем оставшиеся этапы выполнения команды `lw`, а затем рассмотрим выполнение других команд RISC-V.

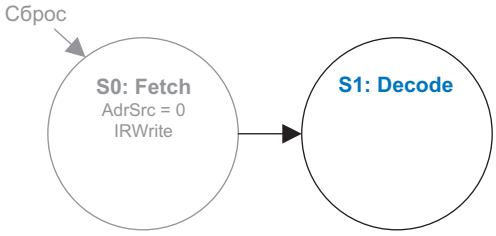


Рис. 7.30 Этап декодирования команды

Вычисление адреса памяти

На третьем этапе команды `lw` процессор вычисляет адрес памяти (MemAdr). АЛУ складывает базовый адрес и смещение, поэтому главный конечный автомат вырабатывает сигналы управления *ALUSrcA* = 10, чтобы прочитать из *SrcA* значение *A* (считываемое из `rs1`), и *ALUSrcB* = 01, чтобы прочитать из *SrcB* значение *ImmExt*. Сигнал *ImmSrc* = 00 вырабатывается дешифратором команд и указывает на необходимость расширения знаком для команд типа *I*, а значения *SrcA* и *SrcB* суммируются в соответствии с сигналом *ALUOp* = 00. В завершение процесса результат АЛУ (т. е. вычисленный адрес) сохраняется в регистре *ALUOut*. На **рис. 7.31** показано состояние главного конечного автомата при вычислении адреса памяти, а на **рис. 7.32** темно-синими линиями показан путь данных в этом состоянии.

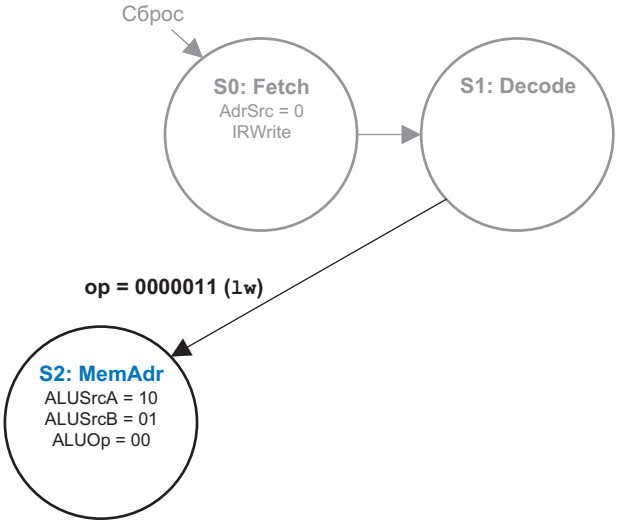


Рис. 7.31 **Вычисление адреса в памяти данных**

Чтение из памяти

Чтобы на этапе чтения из памяти (MemRead) вычисленный адрес из регистра *ALUOut* через мультиплексоры *Result* и *Adr* поступил в адресный порт памяти, главный конечный автомат должен выработать управляющие сигналы *ResultSrc* = 00 и *AdrSrc* = 1. Сигнал *ReadData* принимает значение, прочитанное из памяти по требуемому адресу. В конце этого состояния *ReadData* записывается в регистр данных.

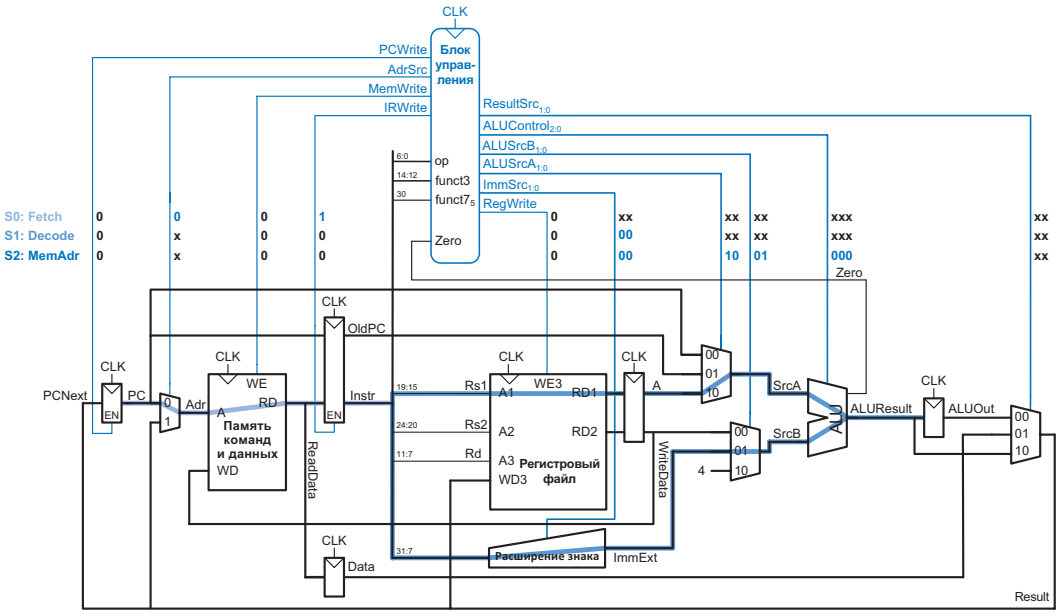


Рис. 7.32 Поток данных во время состояний выборки, декодирования и вычисления адреса

Запись результата

На этапе записи результата (MemWB) данные, считанные из памяти и сохраненные в регистре *Data*, записываются в регистровый файл. В соответствии с управляющим сигналом *ResultSrc* = 01 в качестве источника данных *Data* выбран *Result*, а в соответствии с сигналом *RegWrite* = 1 данные записываются в регистровый файл. Входы адреса регистрового файла и записи данных для третьего порта (*A*3 и *WD*3) уже подключены к `rd` ($Instr_{11:7}$) и *Result* соответственно. На **рис. 7.33** и **7.34** показаны состояния MemRead и MemWB, а также путь данных на обоих этапах. Состояние MemWB – это последний этап выполнения команды `lw`. На **рис. 7.33** также показан переход из состояния MemWB обратно в состояние Fetch, чтобы можно было выбрать следующую команду. При этом счетчик команд пока не инкрементирован. Мы займемся этим дальше.

Перед завершением команды `lw` процессор должен увеличить счетчик команд, чтобы в дальнейшем прочитать следующую команду. Для этого мы могли бы добавить еще одно состояние главного автомата, но внимательный читатель заметит, что АЛУ не используется на этапе выборки, поэтому процессор может сэкономить один такт

Мы начали этот раздел с утверждения, что на каждом этапе можно использовать только один из врямязатратных блоков процессора (память, АЛУ или регистровый файл). Но здесь мы используем как регистровый файл, так и АЛУ. Если есть возможность использовать блоки одновременно и без потери времени на ожидание, то ничто не мешает нам задействовать более одного блока на одном этапе.

и воспользоваться этим состоянием для вычисления $PC+4$ одновременно с извлечением команды. Главный автомат должен сгенерировать следующие управляющие сигналы: $ALUSrcA = 00$, чтобы передать в $SrcA$ старое значение счетчика команд ($OldPC$), $ALUSrcB = 10$ для передачи в $SrcB$ константу 4, $ALUOp = 00$ для выполнения АЛУ сложения $PC+4$. Чтобы записать в PC новое значение, устанавливаются управляющие сигналы $ResultSrc = 10$ ($ALUResult$ в качестве результата) и $PCUpdate = 1$, чтобы принудительно установить $PCWrite$ в единицу (**рис. 7.28**). На **рис. 7.35** показано дополненное состояние Fetch. Остальная часть диаграммы остается такой же, как на **рис. 7.33**. На **рис. 7.36** синим цветом выделен поток данных для вычислений PC + 4. Одновременная выборка команды выделена серым цветом.

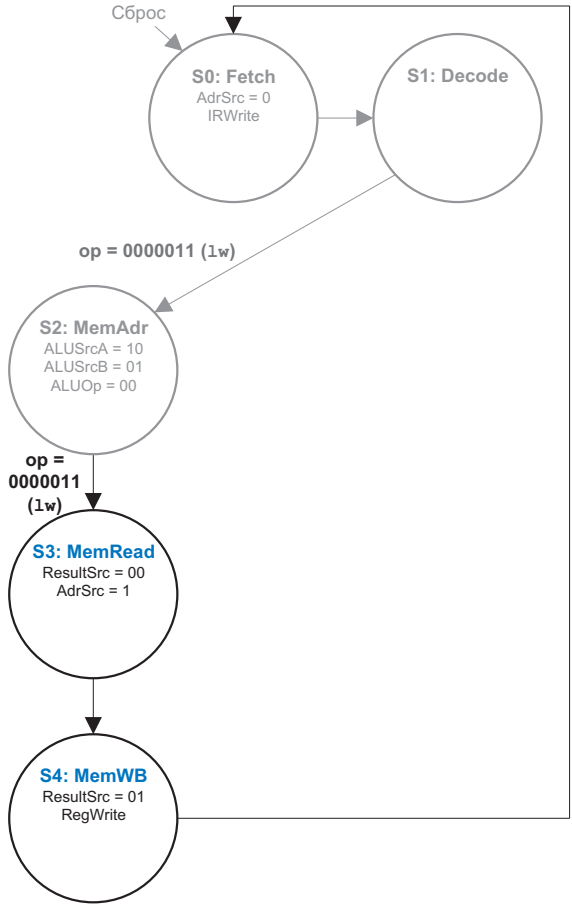


Рис. 7.33 Состояния чтения из памяти (MemRead) и записи в память (MemWB)

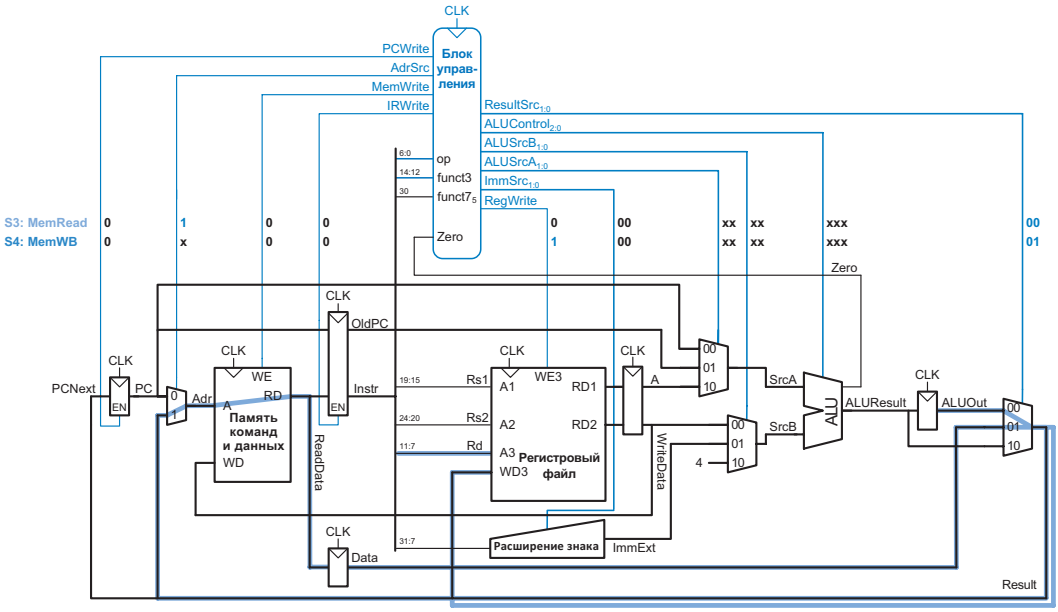


Рис. 7.34 **Поток данных на этапах MemRead и MemWB**

Команда `sw`

Теперь давайте увеличим количество команд RISC-V, которые может обработать управляющий конечный автомат процессора. Все команды обязательно проходят через первые два состояния – выборки и декодирования. Команда `sw` использует такое же состояние вычисления адреса памяти MemAdr, что и `lw`, но затем переходит в состояние записи в память (MemWrite), где полученное из `rs2` значение *WriteData* записывается в память. Сигнал *WriteData* жестко подключен к порту записи данных памяти (*WD*). Адресный порт памяти *Adr* получает вычисленный адрес *ALUOut* в соответствии с управляющими сигналами *ResultSrc* = 00 и *AdrSrc* = 1. Сигнал *MemWrite* = 1 разрешает запись в память. На этом команда `sw` завершается, поэтому главный автомат возвращается в состояние Fetch, чтобы начать следующую инструкцию. На **рис. 7.37** и **7.38** показаны дополненный главный автомат и тракт данных для состояния MemWrite. Первые два состояния конечного автомата (выборка и декодирование), которые не показаны на **рис. 7.37**, такие же, как на **рис. 7.33**.

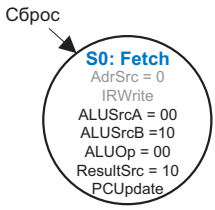


Рис. 7.35 **Увеличение PC в состоянии Fetch**

В состоянии MemAdr значения сигнала *ImmSrc* для команд `lw` и `sw` различаются. Но не забывайте, что сигнал *ImmSrc* генерируется комбинационным декодером команд (рис. 7.28).

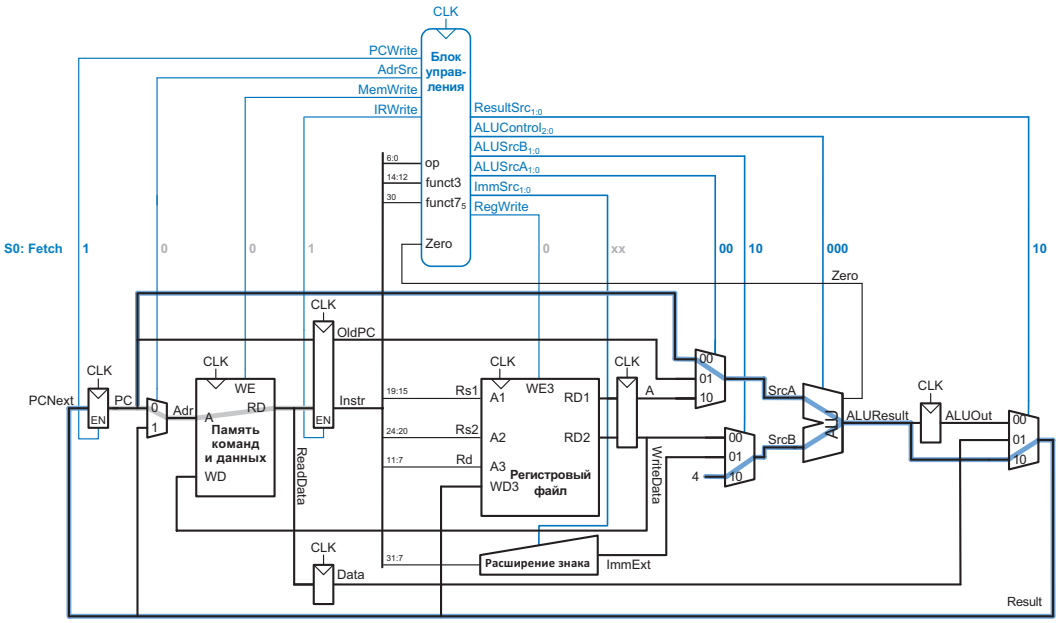


Рис. 7.36 Поток данных при увеличении PC в состоянии Fetch

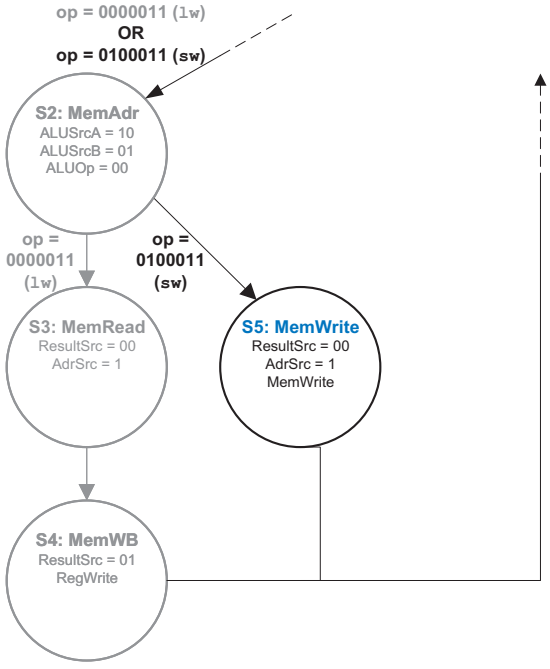


Рис. 7.37 Состояние записи в память (MemWrite)

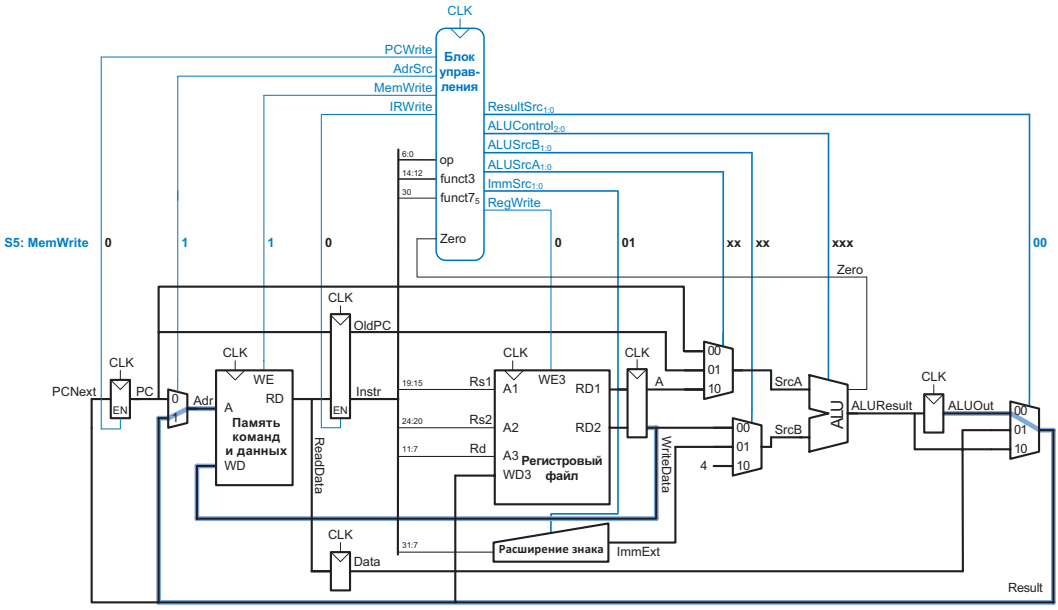


Рис. 7.38 **Поток данных во время состояния записи в память**

Команды типа *R*

После декодирования команды *R*-типа многотактный процессор должен получить результат вычислений в АЛУ и записать его в регистр. Для этого он должен перейти в новое состояние выполнения команды типа *R* (ExecuteR), а именно $ALUSrcA = 10$ и $ALUSrcB = 00$, чтобы прочитать операнд `rs1` в *SrcA* и операнд `rs2` в *SrcB*. Сигнал $ALUOp = 10$, поэтому дешифратор АЛУ использует управляющие поля команды для определения того, какую операцию выполнить.

В конце такта в регистр *ALUOut* записывается результат вычисления *ALUResult*. Затем команды типа *R* переводят процессор в состояние записи результата АЛУ (ALUWB), когда результат вычисления *ALUOut* записывается в регистровый файл.

В состоянии ALUWB генерируются управляющие сигналы $ResultSrc = 00$, чтобы выбрать *ALUOut* в качестве результата и $RegWrite = 1$, чтобы результат был записан в `rd`. На **рис. 7.39** показаны состояния ExecuteR и ALUWB, добавленные в главный автомат. На **рис. 7.40** показан поток данных в обоих состояниях, причем поток данных ExecuteR показан толстыми голубыми линиями, а поток данных ALUWB – толстыми темно-синими линиями.

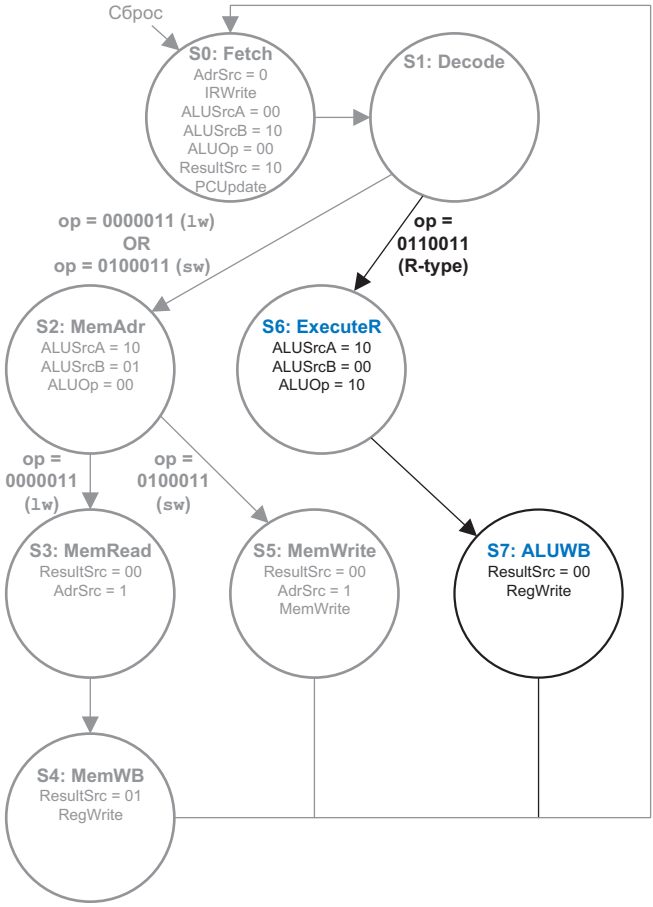


Рис. 7.39 Состояния выполнения вычислений типа *R* (ExecuteR) и обратной записи АЛУ (ALUWB)

Даже если инструкция еще не декодирована в начале состояния декодирования — она может даже и не быть инструкцией `beq`, — целевой адрес перехода все равно вычисляется, как если бы это была команда условного перехода. Если выясняется, что команда не является переходом, или если условие перехода не выполнено, то результат вычисления просто не используется.

Команда `beq`

Для выполнения команды `beq` процессор должен вычислить адрес перехода и сравнить два регистра, чтобы определить, нужно ли перейти по этому адресу. На этапе декодирования команды АЛУ бездействует, поэтому мы можем параллельно воспользоваться им для вычисления целевого адреса условного перехода $OldPC + ImmExt$. Управляющие сигналы $ALUSrcA$ и $ALUSrcB$ равны 01, поэтому в $SrcA$ поступает $OldPC$, а в $SrcB$ — смещение перехода $ImmExt$. Управляющий сигнал $ALUOp = 00$ определяет операцию сложения в АЛУ. В конце состояния декодирования целевой адрес сохраняется в регистре

ALUOut. На **рис. 7.41** показано новое состояние декодирования, а также последующее состояние BEQ, которое мы обсудим дальше. На **рис. 7.42** поток данных в состоянии декодирования обозначен голубыми и серыми линиями. Поток данных при расчете целевого адреса перехода показан линиями голубого цвета, а при чтении регистра и расширении константы – толстыми серыми линиями.

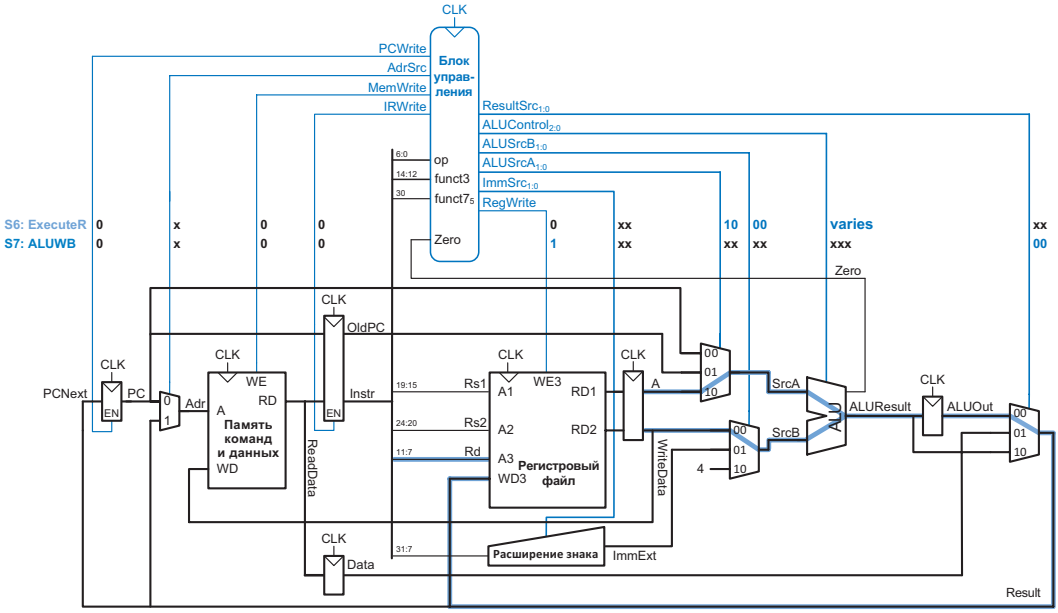


Рис. 7.40 Поток данных в состояниях ExecuteR и ALUWB

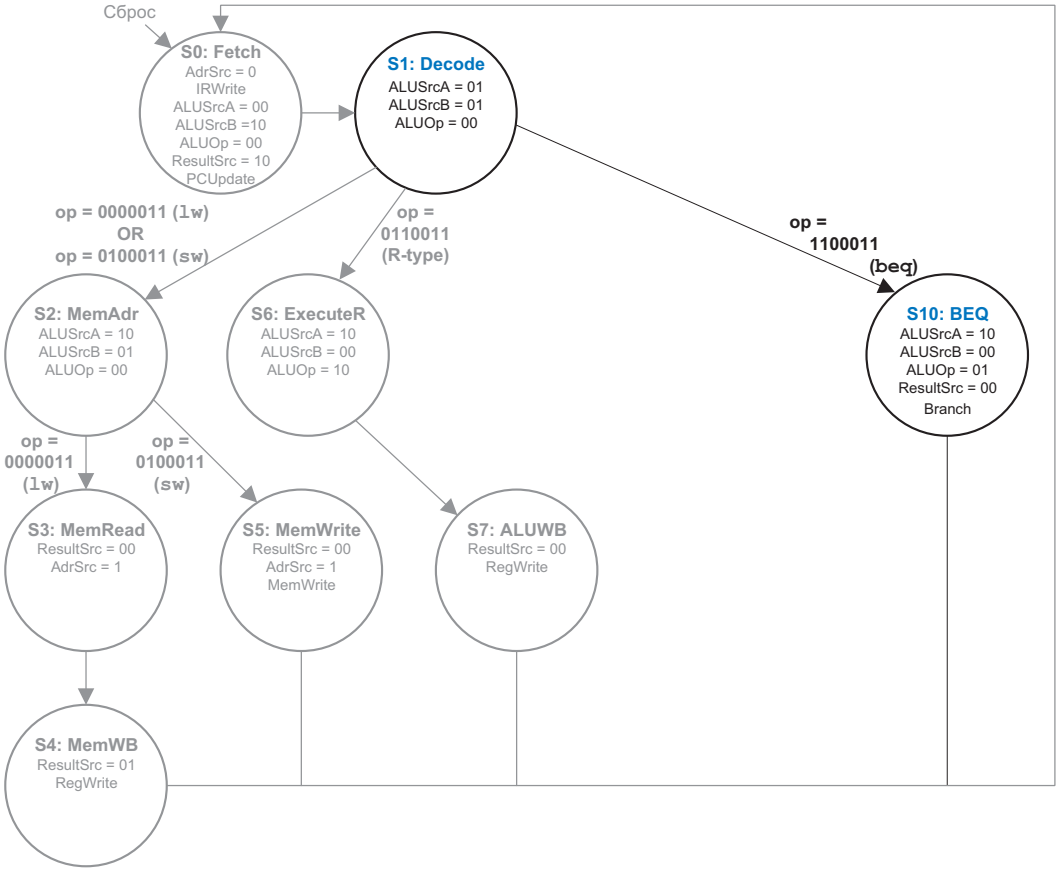


Рис. 7.41 Состояния процессора при выполнении команды `beq`

Из состояния декодирования `beq` процессор переходит в состояние BEQ, где сравнивает исходные регистры. Для выбора значений, считываемых из регистрового файла в *SrcA* и *SrcB*, генерируются управляющие сигналы $ALUSrcA = 10$ и $ALUSrcB = 00$. Сигнал $ALUOp = 01$ указывает АЛУ выполнить вычитание. Если исходные регистры равны, устанавливается нулевой флаг $Zero = 1$ (поскольку `rs1 - rs2 = 0`). При этом генерируется сигнал $Branch = 1$, и если $Zero = 1$, то $PCWrite$ тоже становится равен единице (в соответствии с логикой $PCWrite$ на **рис. 7.28**), а целевой адрес перехода из $ALUOut$ записывается в PC. В соответствии с управляющим сигналом $ResultSrc = 00$ результат из $ALUOut$ сохраняется в регистровый файл. На **рис. 7.41** показано состояние BEQ, а на **рис. 7.42** – поток данных во время состояния BEQ. Поток данных для операции сравнения `rs1` и `rs2` показан линиями темно-синего цвета, а поток данных при записи в PC адреса условного перехода показан се-

рым цветом и проходит через регистр результатов. На этом мы завершаем разработку блока управления для данной команды.

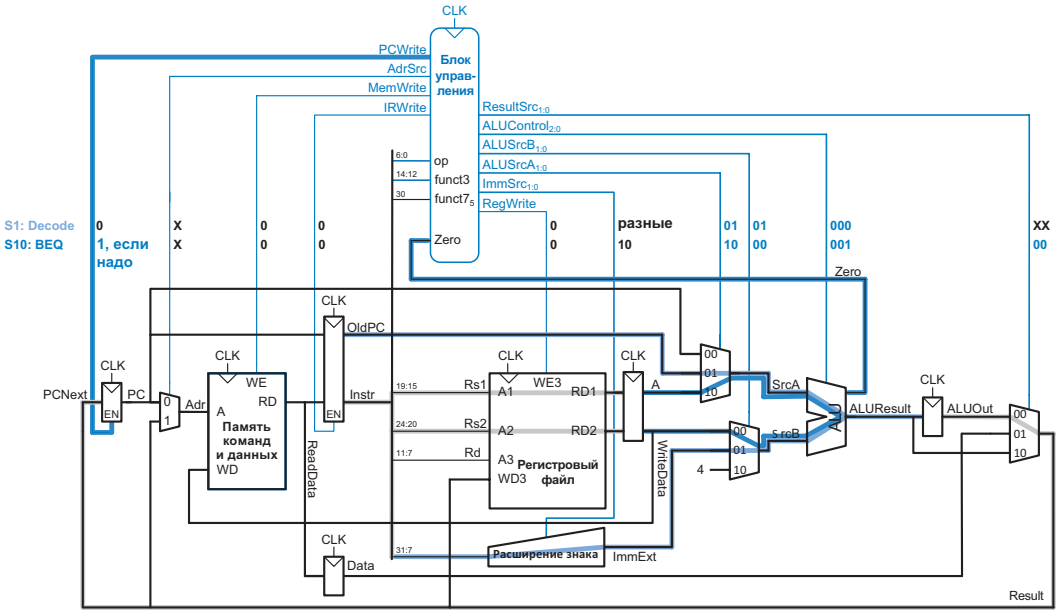


Рис. 7.42 Поток данных во время состояний декодирования и BEQ

7.4.3. Дополнительные команды

Как и в случае с однотактным процессором, далее мы рассмотрим примеры модификации тракта данных и контроллера многотактного процессора для обработки новых команд АЛУ типа *I* (`addi`, `andi`, `ori`, `slti`) и безусловного перехода `jal`.

Пример 7.5 МОДИФИКАЦИЯ МНОГОТАКТНОГО ПРОЦЕССОРА ДЛЯ ПОДДЕРЖКИ КОМАНД АЛУ ТИПА *I*

Модифицируйте многотактный процессор, чтобы добавить поддержку команд АЛУ типа I – `addi`, `andi`, `ori` и `slti`.

Решение Команды типа *I* почти не отличаются от команд типа *R* (`add`, `and`, `or` и `slt`), за исключением того, что второй операнд поступает из *ImmExt*, а не из регистрового файла, поэтому мы вводим новое состояние ExecuteI для выполнения желаемых вычислений для всех инструкций АЛУ типа *I*. Это состояние похоже на ExecuteR, за исключением того, что $ALUSrcB = 01$, чтобы в *SrcB* поступало значение из *ImmExt*. После состояния ExecuteI инструкции типа *I* переходят в состояние записи результата АЛУ (ALUWB) для записи результата в регистровый файл. На **рис. 7.43** показан модифицированный главный автомат, который также включает состояние JAL для **примера 7.6**.

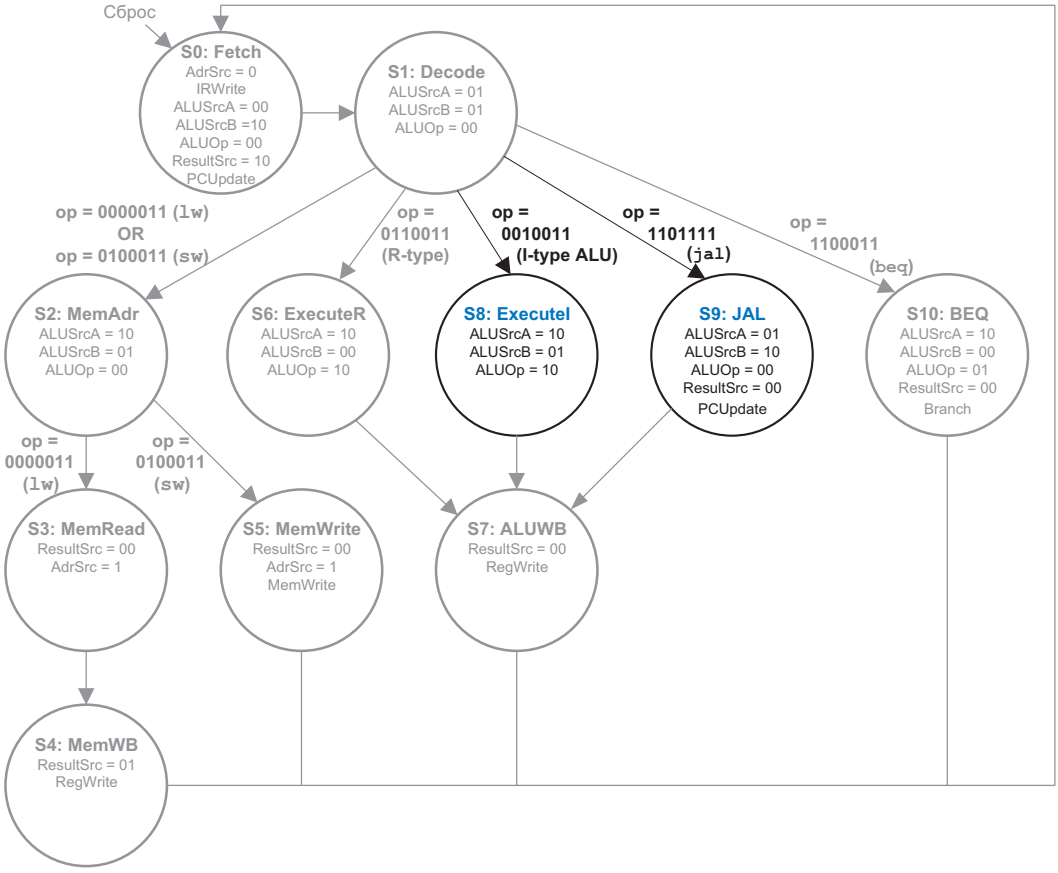


Рис. 7.43 Модифицированный главный автомат с состояниями ExecuteI и JAL

Внимательный читатель мог заметить, что к тому времени, когда процессор достигает состояния JAL, регистр PC уже обновлен значением $PC + 4$. Поэтому мы могли бы просто использовать выход регистра PC для записи в `rd`. Но чтобы использовать содержимое PC, нам пришлось бы расширить мультиплексор источника данных. Приведенное выше решение требует меньше оборудования, поскольку оно использует существующий путь данных.

Пример 7.6 МОДИФИКАЦИЯ МНОГОТАКТНОГО ПРОЦЕССОРА ДЛЯ ПОДДЕРЖКИ КОМАНДЫ `jal`

Модифицируйте многотактный процессор, чтобы добавить поддержку команды `jal`.

Решение Как и инструкции типа *I* из **примера 7.5**, для реализации команды `jal` не требуется никакого дополнительного оборудования. Нужно лишь модифицировать управляющий автомат. Первые два этапа такие же, как и для остальных команд. В состоянии декодирования команды `jal` целевой адрес перехода рассчитывается с использованием того же тракта, что и при расчете целевого адреса условного перехода, но сейчас дешифратор команд генерирует сигнал $ImmSrc = 11$. Поэтому во время этапа декодирования смещение перехода расширяется

знаком и складывается с адресом текущей команды из *OldPC* для формирования целевого адреса перехода, который записывается в регистр *ALUOut* в конце текущего этапа. Затем процессор переходит в состояние JAL, где записывает целевой адрес в счетчик команд и вычисляет адрес возврата (*PC* + 4), чтобы потом записать его в `rd` на следующем этапе. АЛУ вычисляет *OldPC* + 4 в соответствии с управляющими сигналами *ALUSrcA* = 01 (*SrcA* = *OldPC*), *ALUSrcB* = 10 (*SrcB* = 4) и *ALUOp* = 00 (сложение). Чтобы записать целевой адрес перехода в PC, генерируются сигналы *ResultSrc* = 00 (целевой адрес поступает из ALUOut) и *PCUpdate* = 1 (разрешение обновления PC), на основании которого блок управления устанавливает сигнал *PCWrite* = 1. На **рис. 7.43** показано новое состояние JAL, а на **рис. 7.44** – поток данных во время состояния JAL. Путь данных при обновлении PC показан серым цветом, а при вычислении *PC* + 4 – синим. После этапа JAL процессор переходит в состояние ALUWB, где адрес возврата (*ALUOut* = *PC* + 4) записывается в `rd`. На этом выполнение команды `jal` завершается, и управляющий автомат возвращается в состояние Fetch.

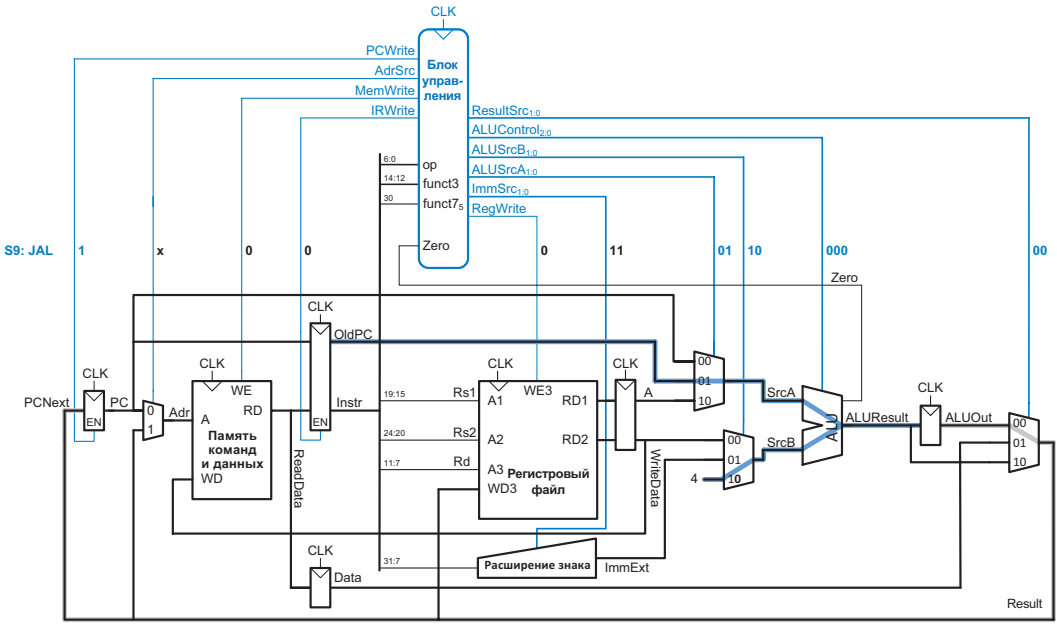
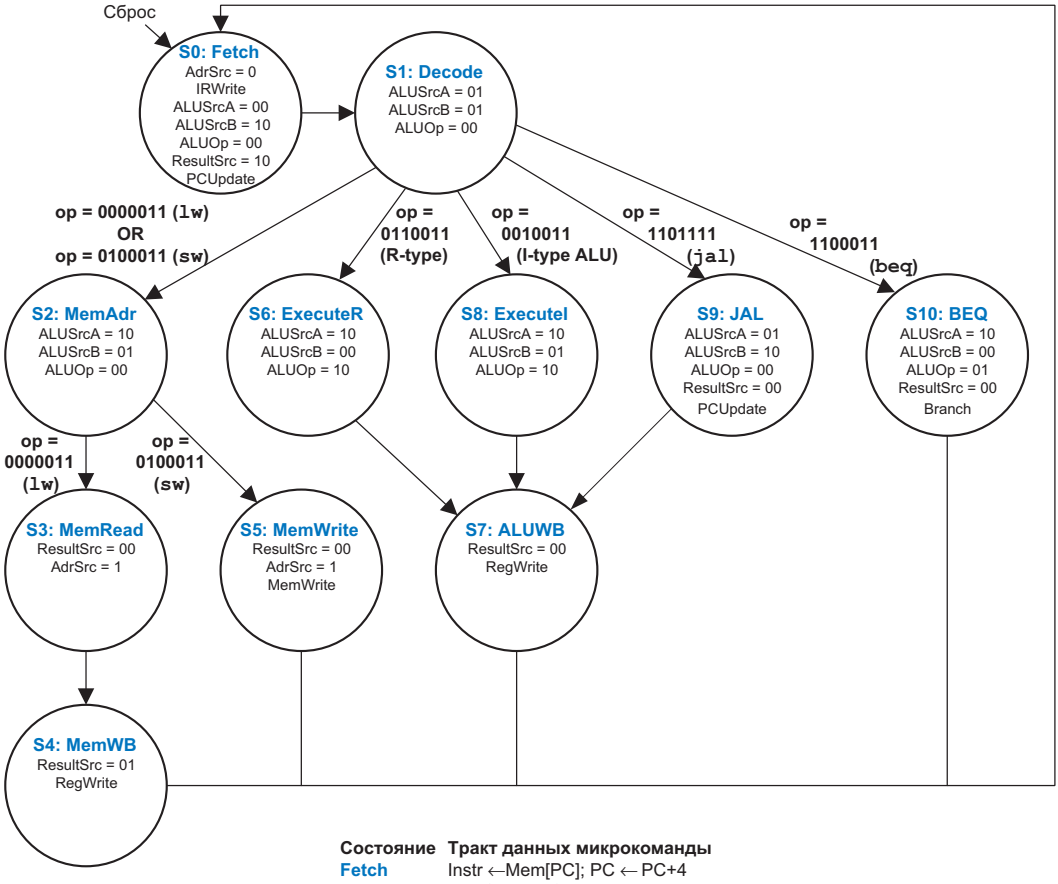


Рис. 7.44 Поток данных во время состояния JAL

На **рис. 7.45** показана полная диаграмма переходов между состояниями управляющего конечного автомата для многотактного процессора. Функции каждого состояния кратко описаны под рисунком. Преобразование такой диаграммы в электрическую схему – простая, но утомительная задача, для решения которой можно использовать методы из **главы 3**. Но лучше описать конечный автомат на языке HDL и синтезировать с использованием методов **главы 4**.



Состояние	Тракт данных микрокоманды
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1oprs2
ExecuteI	ALUOut ← rs1opimm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Рис. 7.45 Полная диаграмма состояний конечного автомата многотактного процессора

7.4.4. Анализ производительности

Время выполнения команды зависит от требуемого количества тактов и их длительности. В отличие от однотактного процессора, который выполняет все команды за один такт, многотактному процессору для выполнения разных команд требуется разное количество тактов. Но поскольку

он выполняет меньшее количество действий за такт, то его такты гораздо короче, чем у однотактного.

Для команд перехода многотактному процессору нужно три такта, для команд типа *I*, типа *R*, перехода и записи в память – четыре, а для команды чтения из памяти – пять. Суммарное количество тактов на команду (CPI) будет зависеть от частоты использования каждой из команд.

Пример 7.7 CPI МНОГОТАКТНОГО ПРОЦЕССОРА

Бенчмарк SPECINT2000 содержит примерно 25 % команд чтения из памяти, 10 % команд записи в память, 11 % команд условного перехода, 2 % команд безусловного перехода и 52 % команд типа R[1]. Определите среднее количество тактов на команду (cycles per instruction, CPI) для этого бенчмарка.

Решение Среднее CPI можно вычислить как взвешенную сумму числа тактов каждой команды, умноженного на долю, которую занимает эта команда в наборе команд программы. Для данного набора среднее CPI = (0,11) (3) + (0,10 + 0,02 + 0,52) (4) + (0,25) (5) = 4,14.

Этот показатель лучше, чем наихудший CPI, равный 5, который мы получили бы, если бы все команды выполнялись за одинаковое количество тактов.

Напомним, что мы разработали многотактный процессор таким образом, чтобы за один такт выполнялась одна операция АЛУ и одна операция доступа к памяти или доступа к регистровому файлу. Предположим, что регистровый файл работает быстрее, чем память, и что запись в память выполняется быстрее, чем чтение из памяти. Изучение тракта данных показывает, что существует два возможных критических пути, которые ограничат минимальную длительность такта, как показано на **рис. 7.46**:

1) путь вычисления *PC* + 4: из регистра PC через мультиплексор SrcA, АЛУ и мультиплексор Result обратно в регистр PC (показан толстыми синими линиями);
2) путь чтения данных из памяти: из регистра ALUOut через мультиплексоры Result и Adr в блок памяти данных и регистр Data (показан толстыми серыми линиями).

Оба этих пути также включают задержку в дешифраторе после обновления состояния (то есть после задержки t_{pcq}), необходимую для выработки управляющих сигналов (выбор мультиплексора и *ALUControl*). Следовательно, длительность такта можно вычислить по формуле **(7.4)**:

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup}. \qquad (7.4)$$

Числовые значения всех элементов выражения будут зависеть от конкретной производственной технологии.

[1] Частотность команд по данным Паттерсона и Хеннесси из книги *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011.

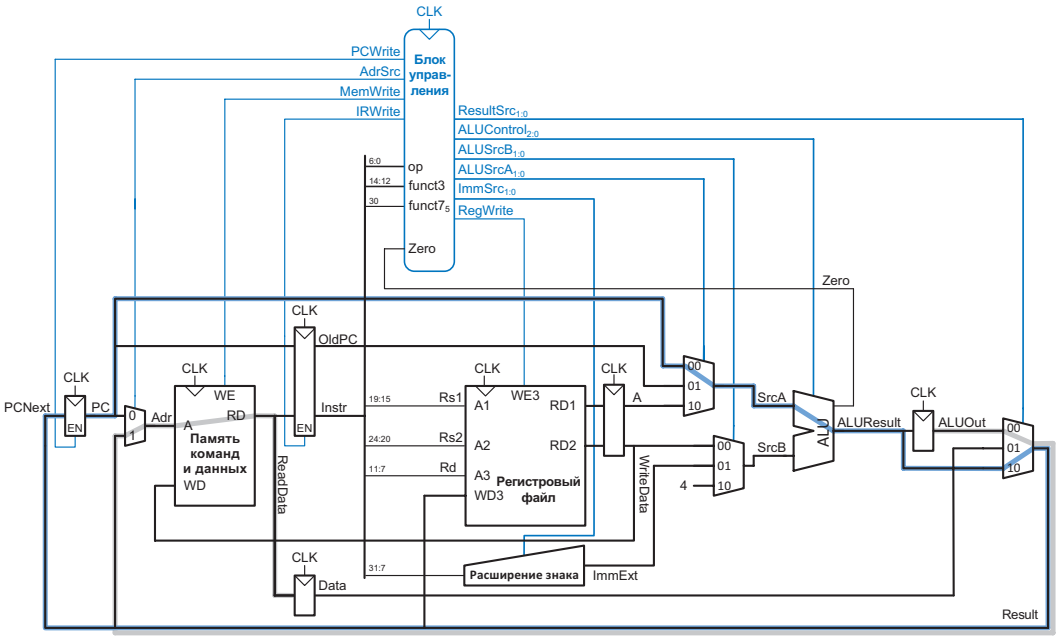


Рис. 7.46 Потенциальные критические пути многотактного процессора

Пример 7.8 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл задумался о том, будет ли многотактный процессор работать быстрее, чем однотактный. В обоих случаях он планирует использовать 7-нм КМОП-техпроцесс с задержками, указанными в **табл. 7.7**. Помогите ему сравнить время выполнения бенчмарка SPECINT2000, состоящего из 100 млрд инструкций, для обоих процессоров (**пример 7.4**).

Решение Согласно уравнению **(7.4)**, длительность такта многотактного процессора равна:

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + t_{mem} + t_{setup} = 40 + 25 + 2\,(30) + 200 + 50 = 375 \text{ пс}.$$

Используя значение CPI 4,14, рассчитанное в **примере 7.7**, находим, что общее время выполнения составляет:

$$T_{multi} = (100\times10^{9} \text{ команд})(4{,}14 \text{ такта}/\text{инструкция})(375\times10^{-12} \text{ с}/\text{такт}) = 155 \text{ с}.$$

Как мы выяснили в **примере 7.4**, у однотактного процессора общее время выполнения программы составило 75 с, следовательно, многотактный процессор работает медленнее.

При разработке многотактного процессора мы хотели избежать того, чтобы все команды выполнялись с той же скоростью, что и самая мед-

ленная. К сожалению, этот пример показывает, что для данных значений CPI и задержек элементов многотактный процессор медленнее, чем однотактный. Фундаментальная проблема оказалась в том, что, несмотря на разбиение самой медленной команды (`lw`) на пять этапов, длительность такта в многотактном процессоре уменьшилась вовсе не в пять раз. Одной из причин явилось то, что не все этапы стали одинаковой длины. Другой причиной стали накладные расходы, связанные с временными регистрами – задержка в 90 пс, равная сумме времени предустановки и времени срабатывания регистра, теперь добавляется не к общему времени выполнения команды, а к каждому этапу по отдельности. Инженеры выяснили, что довольно трудно использовать тот факт, что одни вычисления могут происходить быстрее, чем другие, если разница во времени вычисления мала.

В сравнении с однотактным процессором многотактный процессор, скорее всего, окажется меньшего размера, так как нет необходимости в двух дополнительных сумматорах, а память команд и данных объединена в один блок. Взамен ему требуется пять неархитектурных (временных) регистров и несколько дополнительных мультиплексоров.

7.5. Конвейерный процессор

Напомним, что пропускная способность – это количество задач (в данном случае команд), выполняемых за секунду. Задержка – это время, необходимое для выполнения данной команды от начала до конца (раздел 3.6).

Конвейеризация, о которой мы говорили в **разделе 3.6**, – это мощное средство увеличения пропускной способности цифровой системы. Мы разработаем конвейерный процессор, разделив однотактный процессор на пять стадий. Таким образом, пять команд смогут выполняться одновременно, по одной в каждой из стадий. Так как каждая стадия содержит только одну пятую от всей логики процессора, то частота тактового сигнала может быть почти в пять раз выше. В идеальном случае латентность команд не изменится, а пропускная способность вырастет в пять раз. Микропроцессоры выполняют миллионы и миллиарды инструкций в секунду, так что производительность важнее, чем латентность. Конвейеризация требует определенных накладных расходов, так что в реальной жизни пропускная способность будет ниже, чем в идеальном случае, но в любом случае у конвейеризации так много преимуществ, а обходится она так дешево, что все современные высокопроизводительные микропроцессоры – конвейерные.

Чтение и запись в память и регистровый файл, а также использование АЛУ обычно составляют наибольшие задержки в процессоре. Мы поделим конвейер на стадии таким образом, чтобы каждая из них включала ровно одну из этих операций. Стадии мы назовем *Fetch* (Выборка), *Decode* (Декодирование), *Execute* (Выполнение), *Memory* (Доступ к па-

мяти) и *Writeback* (Запись результатов). Они похожи на пять этапов выполнения команды `lw` в многотактном процессоре. В стадии *Fetch* процессор читает команду из памяти команд. В стадии *Decode* процессор читает операнды из регистрового файла и дешифрует команду, чтобы установить управляющие сигналы. В стадии *Execute* процессор выполняет вычисления в АЛУ. В стадии *Memory* процессор читает или пишет в память данных. Наконец, в стадии *Writeback* процессор, если нужно, записывает результат в регистровый файл.

На **рис. 7.47** приведена временная диаграмма для сравнения однотактного и конвейерного процессоров. По горизонтальной оси отложено время, а по вертикальной – команды. Значения задержек логических элементов на диаграмме взяты из **табл. 7.7**, но для простоты мы пренебрегаем задержками мультиплексоров и регистров. В однотактном процессоре, показанном на **рис. 7.47 (a)**, первая команда выбирается из памяти в момент времени 0, далее процессор читает операнды из регистрового файла, после чего АЛУ выполняет необходимые вычисления. Потом происходит доступ к памяти данных, и результат записывается в регистровый файл через 680 пс. Выполнение второй команды начинается после того, как закончена первая. Таким образом, как следует из диаграммы, однотактный процессор обеспечивает латентность команд, равную 200 + 100 + 120 + 200 + 60 = 680 пс (**табл. 7.7**), и пропускную способность, равную одной команде за 680 пс (1,47 млрд команд в секунду).

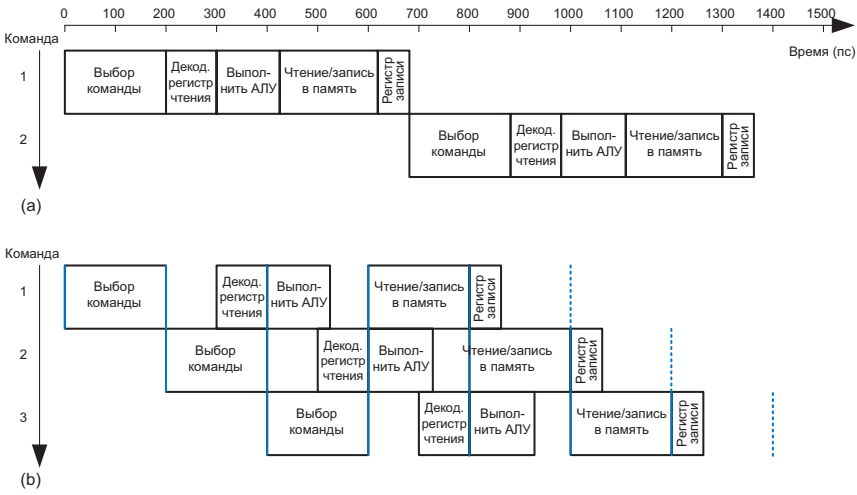


Рис. 7.47 Временная диаграмма однотактного (a) и конвейерного (b) процессоров

В конвейерном процессоре, приведенном на **рис. 7.47 (b)**, длина стадии равна 200 пс и определяется самой медленной стадией – стадией доступа к памяти (Fetch или Memory). Каждая стадия конвейера обозначе-

на сплошными или пунктирными вертикальными синими линиями. В начальный момент времени первая команда выбирается из памяти. Через 200 пс первая команда попадает в стадию *Decode*, а вторая команда выбирается из памяти. Когда прошло 400 пс, первая команда начинает выполняться, вторая попадает в стадию *Decode*, а третья выбирается. И так далее, пока все команды не завершатся. Латентность команд составляет 5×200 = 1000 пс. Латентность команд в конвейерном процессоре немного больше, чем в однотактном, потому что стадии конвейера не идеально сбалансированы, то есть не содержат абсолютно одинаковое количество логики. Пропускная способность – одна команда за 200 пс (5 млрд инструкций в секунду), то есть за каждый такт завершается одна команда. Эта пропускная способность в 3,4 раза больше, чем у однотактного процессора, – хоть и не в пять раз больше, но тем не менее это значительное увеличение пропускной способности.

Не забывайте, что в этом абстрактном сравнении производительности однотактного и конвейерного процессоров мы игнорируем накладные расходы, связанные с задержками дешифратора, мультиплексора и регистров.

На **рис. 7.48** показано абстрактное представление работающего конвейера, демонстрирующее продвижение команд по конвейеру. Каждая стадия изображена символом, содержащим главный компонент стадии – память команд (instruction memory, IM), чтение регистрового файла (register file, RF), АЛУ, память данных (DM) и запись в регистровый файл (writeback). Если смотреть на строки, то можно узнать, на каком такте команда находится в той или иной стадии. Например, команда `sub` выбирается из памяти на третьем такте и выполняется на пятом. Если смотреть на столбцы, то можно узнать, чем заняты стадии конвейера на конкретном такте. Например, на шестом такте регистровый файл записывает сумму в `s3`, память данных бездействует, АЛУ выполняет вычисления (`s11 & t0`), регистр `t4` читается из регистрового файла, а инструкция `or` выбирается из памяти команд. Стадии закрашены серым цветом, если они используются. Например, память данных используется командами `lw` на четвертом такте и `sw` на восьмом. Память команд и АЛУ используются на каждом такте. Результат записывается в регистровый файл всеми командами, кроме `sw`. В конвейерном процессоре значения записываются в регистровый файл в первой части такта, а читаются во второй, что также отмечено на рисунке. В этом случае данные могут быть записаны и затем прочитаны обратно за один и тот же такт.

Главная проблема в конвейерных системах – разрешение *конфликтов* (hazards), которые возникают, когда результаты одной из команд требуются для выполнения следующей команды до того, как предыдущая завершится. Например, если бы команда `add` на **рис. 7.48** использовала `s2` в качестве источника вместо `s10`, то возник бы конфликт, потому что регистр `s2` еще не был бы записан командой `lw` в тот момент, когда команда `add` должна была его прочитать в третьем такте. В этом разделе мы рассмотрим пересылку данных через *байпас* (bypassing или forwarding),

приостановку конвейера (stall) и *сброс конвейера* (flush) в качестве методов разрешения конфликтов. После этого повторно оценим производительность, приняв во внимание накладные расходы на организацию *конвейерной обработки* (sequencing) и влияние на нее конфликтов.

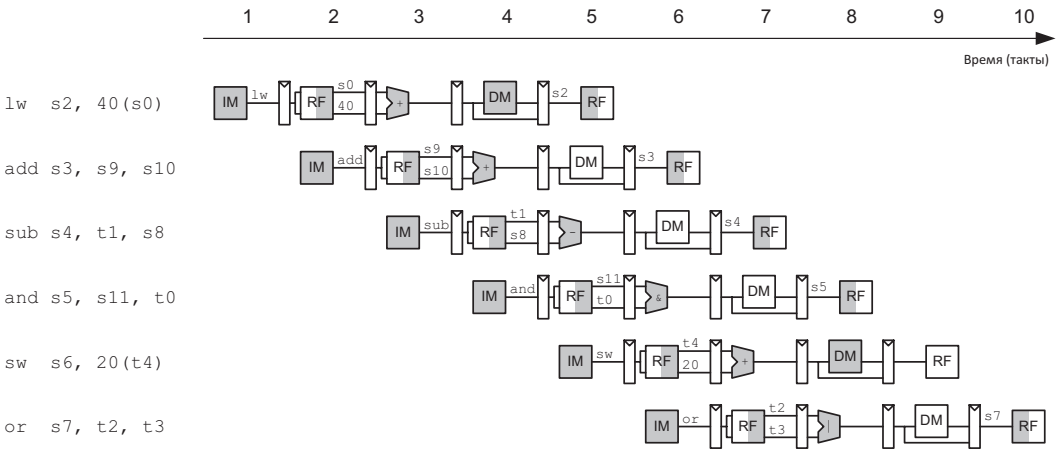


Рис. 7.48 Абстрактное представление работающего конвейера

7.5.1. Конвейерный тракт данных

Конвейерный тракт данных можно получить, разделив однотактный тракт данных на пять стадий, отделенных друг от друга регистрами (pipeline registers). На **рис. 7.49 (a)** показан однотактный тракт данных, растянутый таким образом, чтобы оставить место для регистров между стадиями. На **рис. 7.49 (b)** показан конвейерный тракт данных, поделенный на пять стадий путем вставки в него четырех регистров. Названия стадий и границы между ними показаны синим цветом. Ко всем сигналам добавлен суффикс (F, D, E, M или W), показывающий, к какой стадии они относятся.

Регистровый файл особенный в том смысле, что процессор читает из него в стадии Decode, а пишет в стадии Writeback. Поэтому, несмотря на то что на рисунке он находится в стадии Decode, адрес и данные для записи приходят из стадии Writeback. Эта обратная связь будет приводить к конфликтам конвейера, которые мы рассмотрим в **разделе 7.5.3**. В конвейерном процессоре значения записываются в регистровый файл по заднему фронту тактового сигнала CLK, чтобы он мог записать результат в первой половине такта и прочитать этот результат во второй половине такта для использования в следующей команде.

Одна маленькая, но чрезвычайно важная проблема организации конвейерной обработки данных – это то, что все сигналы, относящиеся к конкретной команде, должны обязательно продвигаться по конвейеру

одновременно друг с другом, в унисон. На **рис. 7.49 (b)** есть связанная с этим ошибка. Можете ли вы ее найти?

Ошибка – в логике записи в регистровый файл, которая происходит в стадии *Writeback*. В регистровый файл записывается значение *ResultW* из стадии *Writeback*. Но номер регистра назначения поступает из `RdD` (*InstrD*$_{11:7}$) из стадии *Decode*. На конвейерной диаграмме, приведенной на **рис. 7.48**, на пятом такте результат команды `lw` будет ошибочно записан в регистр `s5`, а не в `s2`.

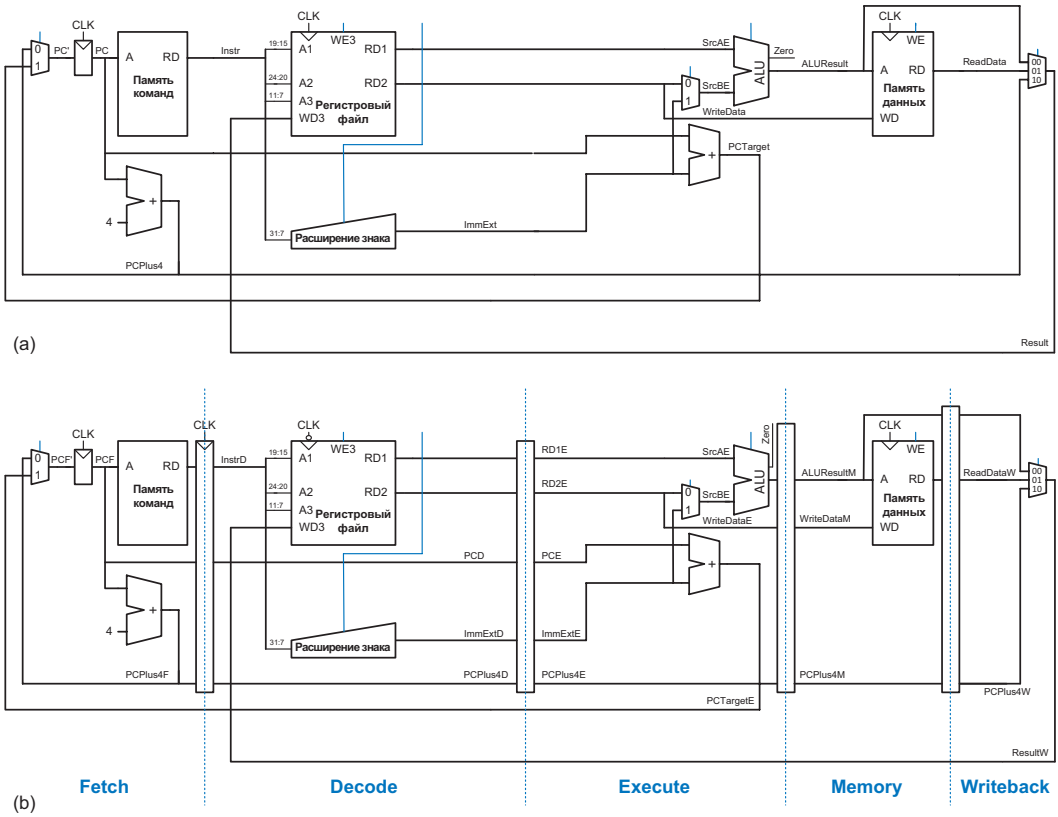


Рис. 7.49 Однотактный (a) и конвейерный (b) тракты данных

На **рис. 7.50** показан исправленный тракт данных, помеченный синим цветом. Сигнал *Rd* теперь конвейерно проходит через этапы *Execution*, *Memory* и *Wruteback*, поэтому он остается синхронным с остальными сигналами команды. Теперь *RdW* и *ResultW* подаются на входы регистрового файла в стадии Writeback одновременно.

Проницательный читатель может заметить, что в логике создания *PCF'* (следующего значения PC) тоже есть проблема, потому что этот сигнал может понадобиться изменить одновременно в стадиях *Fetch* или

Execute (используя сигналы *PCPlus4F* или *PCTargetE* соответственно). В **разделе 7.5.3** мы покажем, как разрешить данный конфликт.

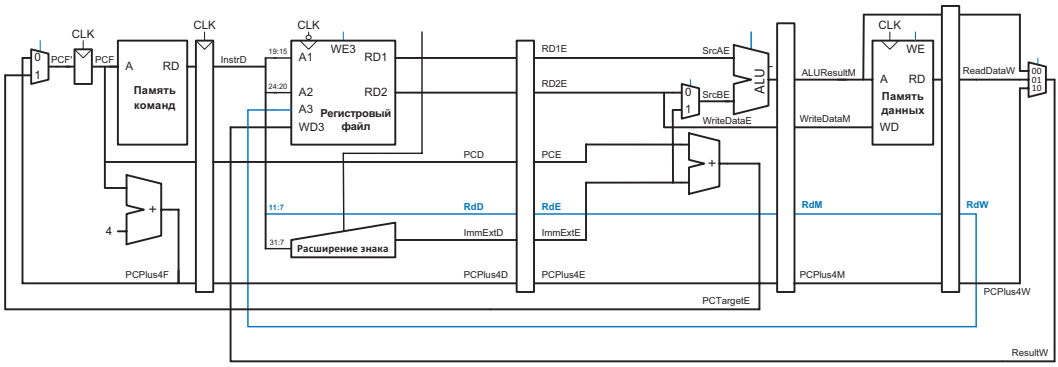


Рис. 7.50 Исправленный тракт данных

7.5.2. Конвейерное устройство управления

Конвейерный процессор использует те же управляющие сигналы, что и однотактный процессор, поэтому применяет аналогичное устройство управления. В стадии *Decode* оно, в зависимости от полей `op`, `funct3` и `funct75` команды, формирует управляющие сигналы, как было показано в **разделе 7.3.3** для однотактного процессора. Эти управляющие сигналы должны быть конвейеризированы точно так же, как и тракт данных, чтобы оставаться синхронными с командой, перемещающейся из одной стадии в другую.

Полностью конвейерный процессор с устройством управления показан на **рис. 7.51**. Аналогично сигналу *Rd* на **рис. 7.50**, сигнал *RegWrite*, пройдя через несколько регистров, обязательно должен дойти до стадии Writeback, перед тем как попасть на вход регистрового файла. Помимо команд АЛУ типа *R*, `lw`, `sw` и `beq`, этот конвейерный процессор также поддерживает команды типа *I* и `jal`.

7.5.3. Конфликты

В конвейерном процессоре выполняется несколько команд одновременно. Когда одна из них зависит от результатов другой, еще не завершенной команды, то говорят, что произошел *конфликт* (hazard) в конвейере. Процессор может читать и записывать в регистровый файл за один такт. Запись происходит в первой части такта, а чтение – во второй, так что значение в регистр можно записать и затем прочитать обратно за один такт, и это не приведет к конфликту.

На **рис. 7.52** показан конфликт, который возникает, когда одна команда пишет в регистр (`s8`), а следующая команда читает из него. Ре-

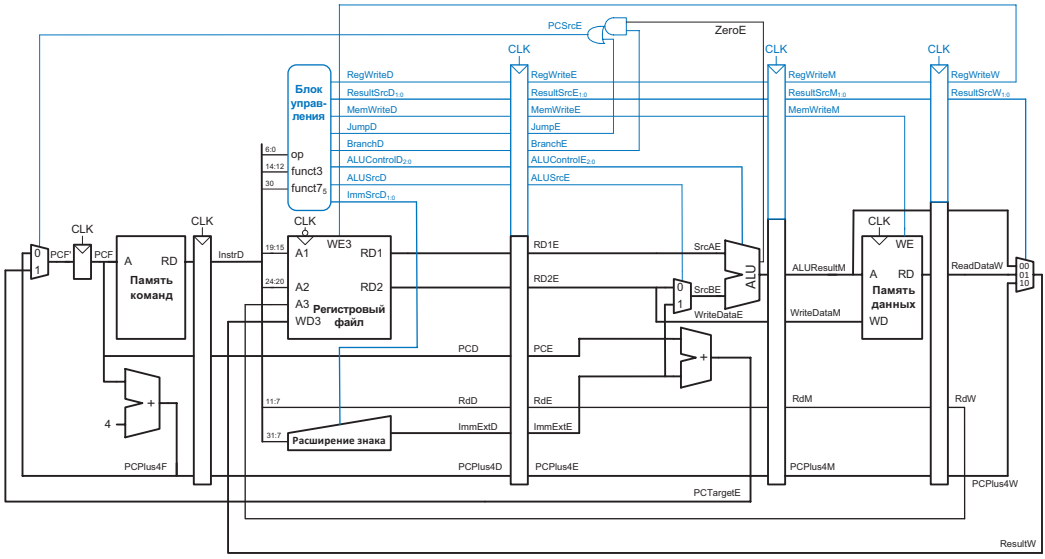


Рис. 7.51 **Конвейерный процессор с устройством управления**

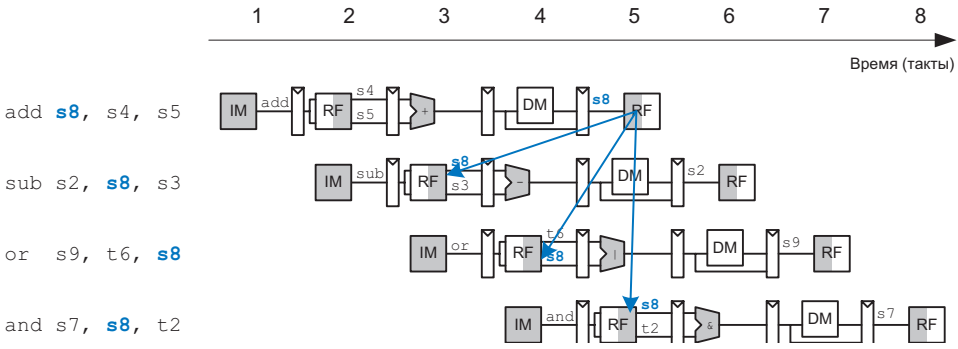


Рис. 7.52 **Абстрактная схема конвейера, демонстрирующая конфликты**

гистр `s8` записывается в регистровый файл в такте 5, а синие стрелки показывают, когда на самом деле должна была произойти запись для использования в следующих командах. Это называется конфликтом *чтения после записи* (read after write, RAW). Команда `add` записывает результат в `s8` в первой половине пятого такта. Но команда `sub` читает `s8` на третьем такте, то есть получает неверное значение. Команда `or` читает `s8` на четвертом такте и тоже получает неверное значение. Команда `and` читает `s8` во второй половине пятого такта и наконец-то получает корректное значение, которое было записано в первой половине пятого такта. Последующие команды также прочитают правильное значение из `s8`. Как следует из диаграммы, конфликт в конвейере возникает тогда,

когда команда записывает значение в регистр и хотя бы одна из следующих двух команд читает его. Если не принять мер, конвейер вычислит неправильный результат.

В качестве программного решения программист или компилятор мог бы вставить пустые команды `nop` между командами `add` и `sub`, чтобы зависимая команда не читала результат (`s8`) до тех пор, пока он не будет доступен в регистровом файле, как показано на **рис. 7.53**. Такая программная блокировка усложняет программирование и снижает производительность, поэтому ее нельзя назвать удачным решением проблемы.

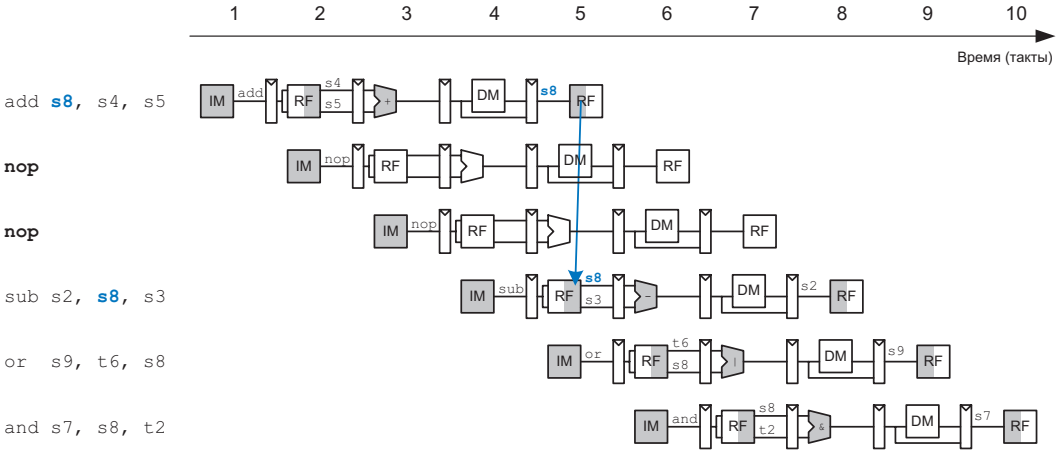


Рис. 7.53 Устранение конфликта данных с помощью команд `nop`

При более внимательном рассмотрении **рис. 7.52** оказывается, что результат команды `add` вычисляется в АЛУ на третьем такте, а команде `and` он требуется лишь на четвертом. В принципе, мы могли бы переслать результат выполнения первой команды второй до того, как он будет записан в регистровый файл, разрешив конфликт чтения после записи без необходимости приостанавливать конвейер. В некоторых других случаях, которые мы рассмотрим далее, конвейер все-таки придется приостанавливать, чтобы дать процессору время вычислить требуемый результат до того, как он понадобится последующим командам. В любом случае, чтобы программы выполнялась корректно, несмотря на конвейеризацию, мы должны что-то предпринять для разрешения конфликтов.

Конфликты можно разделить на *конфликты данных* (data hazards) и *конфликты управления* (control hazards). Конфликт данных происходит, когда команда пытается прочитать из регистра значение, которое еще не было записано предыдущей командой. Конфликт управления происходит, когда процессор выбирает из памяти следующую команду до того, как стало ясно, какую именно команду надо выбрать. В оставшейся части этого раздела мы добавим в процессор *блок разрешения конфлик-*

тов (hazard unit), который будет выявлять и разрешать конфликты таким образом, чтобы процессор выполнял программы корректно.

Разрешение конфликтов пересылкой через байпас

Некоторые конфликты данных можно разрешить путем пересылки результата через байпас из стадий Memory или Writeback в ожидающую этот результат команду, находящуюся в стадии Execute. Чтобы организовать байпас, понадобится добавить мультиплексоры перед АЛУ. Теперь операнд можно получить либо из регистрового файла, либо напрямую из стадий Memory или Writeback, как показано на **рис. 7.54**. Программа, которую мы рассматриваем в качестве примера, вычисляет значение в `s8` с помощью команды `add`, а затем использует сохраненное значение в трех последующих командах. Таким образом, на четвертом такте `s8` пересылается через байпас из стадии *Memory*, где находится команда `add`, в стадию *Execute*, где находится команда `sub`, которой нужен результат выполнения `add`. На пятом такте `s8` пересылается из стадии *Writeback*, где теперь находится команда `add`, в стадию *Execute*, где находится ожидающая ее результата команда `or`. С другой стороны, для инструкции `and` пересылка не требуется, потому что `s8` записывается в регистровый файл в первой половине пятого такта и считывается во второй половине.

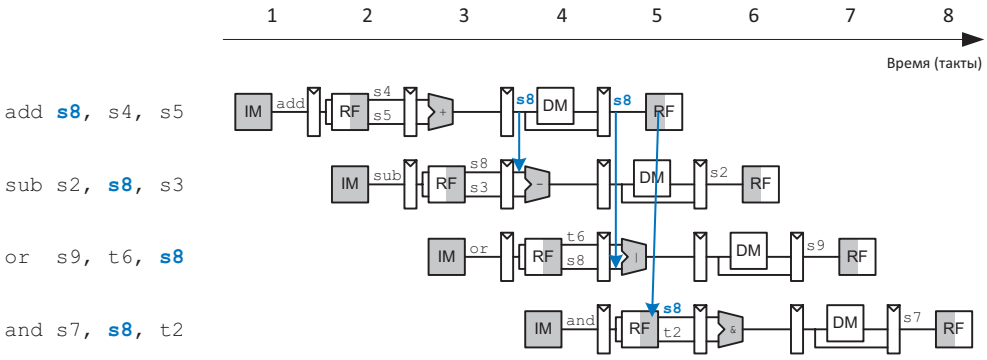


Рис. 7.54 Пересылка данных через байпас

Пересылка данных через байпас необходима, если номер любого из регистров операндов команды, находящейся в стадии *Execute*, равен номеру регистра результата команды, находящейся в стадии *Memory* или *Writeback*. На **рис. 7.55** показан модифицированный конвейерный процессор с байпасом. У него есть блок обнаружения конфликтов и два новых мультиплексора. Блок обнаружения конфликтов получает на свой вход номера регистров, хранящих операнды команды, находящейся в стадии *Execute* (*Rs1E* и *Rs2E*), а также номера регистров результатов команд, находящихся в стадиях *Memory* и *Writeback* (*RdM* и *RdW*). Еще ему необходимы сигналы *RegWrite* из стадий *Memory* и *Writeback* (*Reg-*

WriteM и *RegWriteW*). Эти сигналы показывают, нужно ли на самом деле записывать результат в регистр (например, команды `sw` и `beq` не записывают свои результаты в регистровый файл, поэтому их результаты пересылать не нужно).

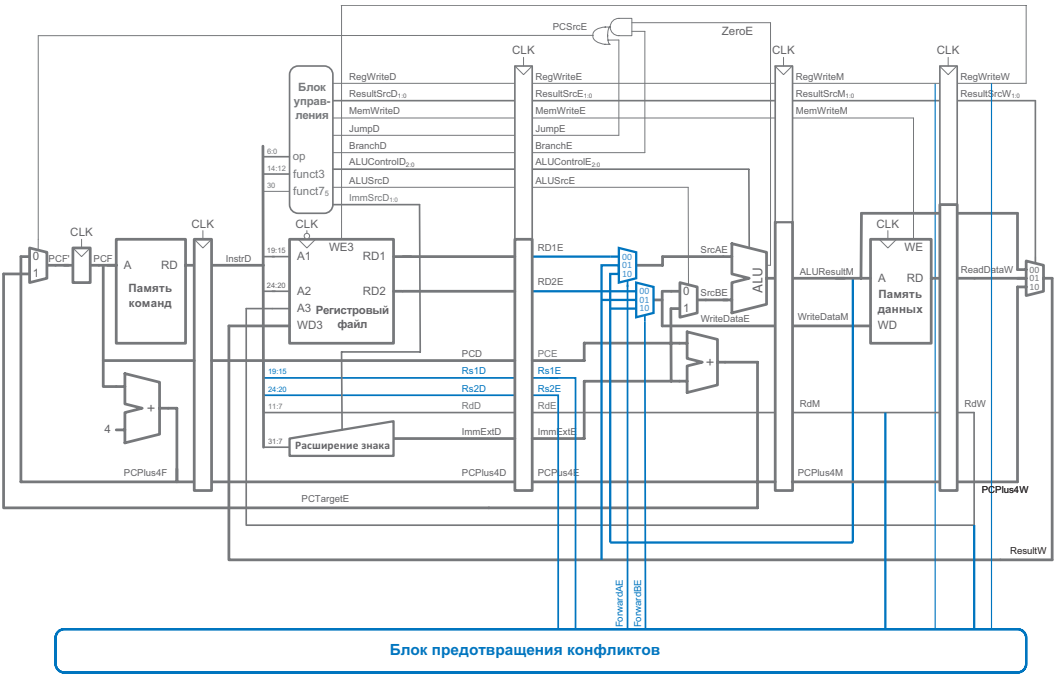


Рис. 7.55 Разрешение конфликтов данных в конвейере при помощи байпаса

Блок обнаружения конфликтов управляет мультиплексорами байпаса (forwarding multiplexers), которые определяют, взять ли операнды из регистрового файла или переслать их напрямую из стадии *Memory* или *Writeback* (*ALUResultM* или *ResultW*). Если в одной из этих стадий происходит запись результата в регистр и номер этого регистра совпадает с номером регистра операнда следующей команды, то используется байпас. Регистр `x0` – исключение, он всегда содержит ноль, поэтому его нельзя пересылать. Если номера регистров результатов в стадиях *Memory* и *Writeback* одинаковы, то приоритет отдается стадии *Memory*, так как она содержит более новую команду. Ниже приведена функция, определяющая логику пересылки данных в операнд *SrcAE* (*ForwardAE*). Логика пересылки данных *SrcBE* (*ForwardBE*) точно такая же, за исключением того, что она проверяет поле *Rs2E* вместо *Rs1E*.

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then  // пересылка
                                                                из стадии Memory
        ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then  // пересылка
                                                                из стадии Writeback
        ForwardAE = 01
else    ForwardAE = 00                                          // без пересылки
```

Разрешение конфликтов данных приостановками конвейера

Пересылка данных через байпас может разрешить конфликт при чтении после записи, только если результат вычисляется в стадии *Execute*, потому что лишь в этом случае его можно сразу переслать в стадию *Execute* следующей команды. К сожалению, команда `lw` не может прочитать данные раньше, чем в конце стадии *Memory*, поэтому ее результат нельзя переслать в стадию *Execute* следующей команды. В этом случае мы будем говорить, что латентность команды `lw` равна двум тактам, потому что зависимая команда не может использовать результат `lw` раньше, чем через два такта. Эта проблема показана на **рис. 7.56**. Команда `lw` получает данные из памяти в конце четвертого такта. Но команде `and` эти данные требуются в качестве операнда уже в самом начале четвертого такта. Пересылка данных через байпас не поможет разрешить этот конфликт.

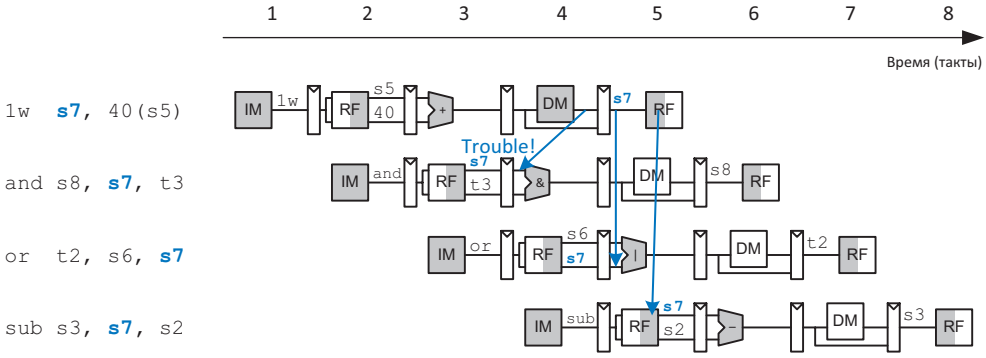


Рис. 7.56 Проблема при пересылке результата команды `lw` через байпас

Альтернативное решение – приостановить конвейер, задержав все операции до тех пор, пока данные не станут доступны. На **рис. 7.57** показана приостановка зависимой команды (`and`) в стадии *Decode*. Команда `and` попадает в эту стадию на третьем такте и остается там на четвертом такте. Следующая команда (`or`) должна, соответственно, оставаться в стадии *Fetch* в течение третьего и четвертого тактов, так как стадия *Decode* занята.

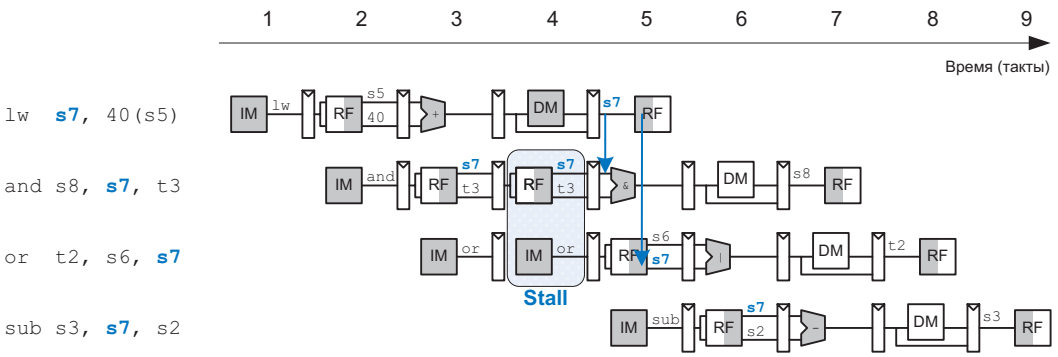


Рис. 7.57 Разрешение конфликта приостановкой конвейера

На пятом такте результат команды `lw` можно через байпас переслать из стадии *Writeback* в стадию *Execute*, где будет находиться команда `and`. На этом же такте операнд `s7` команды `or` может быть прочитан прямо из регистрового файла, без какой-либо пересылки данных.

Заметьте, что теперь стадия *Execute* на четвертом такте не используется. Аналогично стадия *Memory* не используется на пятом такте, а *Writeback* – на шестом. Эта неиспользуемая стадия, проходящая по конвейеру, называется *пузырьком* (bubble) и ведет себя так же, как команда `nop`. Пузырек получается путем обнуления всех управляющих сигналов стадии *Execute* на время приостановки стадии *Decode*, так что он не приводит ни к каким изменениям архитектурного состояния.

Таким образом, стадию конвейера можно приостановить, если запретить обновление регистра, находящегося между этой и предыдущей стадиями. Как только какая-либо стадия приостановлена, все предыдущие стадии тоже должны быть приостановлены, чтобы ни одна из команд не пропала. Регистр, находящийся сразу после приостановленной стадии, должен быть очищен, чтобы «мусор» не попал в конвейер. Приостановки конвейера ухудшают производительность, поэтому должны использоваться, только если по-другому разрешить конфликт данных нельзя.

На **рис. 7.58** показан модифицированный процессор, который умеет приостанавливать конвейер для разрешения конфликтов данных, возникающих при выполнении команды `lw`. Для того чтобы опасная команда остановила конвейер, должны быть соблюдены следующие условия:

1) загрузка слова находится на стадии *Execute* (обозначено как *ResultSrcE0* = 1);
2) номер регистра результата (*RdE*) совпадает с *Rs1D* или *Rs2D*, исходными операндами команды, находящейся на стадии *Decode*.

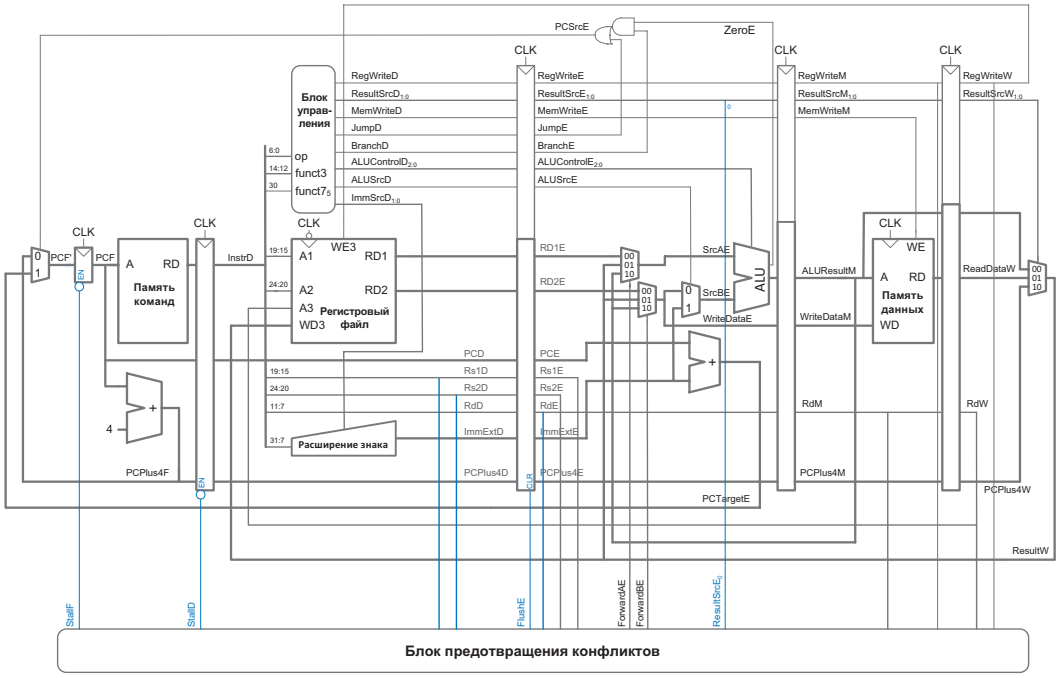


Рис. 7.58 Разрешение конфликтов в конвейере при помощи приостановок

Для приостановки стадий *Fetch* и *Decode* нужно добавить вход разрешения работы (EN) временным регистрам, расположенным перед этими стадиями, а также вход синхронного сброса (CLR) временному регистру, расположенному перед стадией *Execute*. Когда возникает необходимость приостановить конвейер из-за команды `lw`, устанавливаются сигналы *StallD* и *StallF*, запрещающие временным регистрам перед стадиями *Decode* и *Fetch* изменять их старое значение. Также устанавливается сигнал *FlushE*, очищающий содержимое временного регистра перед стадией *Execute*, что приводит к появлению пузырька. Сигнал *lwStall* (приостановка по команде `lw`) модуля конфликтов указывает, когда конвейер должен быть приостановлен в ожидании выполнения команды. Всякий раз, когда *lwStall* равен единице, устанавливаются все сигналы *приостановки* (stall) и *очистки* (flush). Таким образом, логика формирования сигналов приостановки и очистки выглядит так:

$$lwStall = ResultSrcE0 \;\&\; ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF = StallD = FlushE = lwStall$$

Описанная здесь логика формирования сигнала *lwStall* может вызвать ненужную приостановку процессора, когда регистром назначения является `x0` или когда возникает *ложная зависимость*, — ситуация, когда на стадии *Decode* оказывается команда типа *J* или *I*, которая случайным образом вызывает ложное совпадение между битами в поле константы и *RdE*. Следует при этом отметить, что эти случаи редки, и они вызывают лишь небольшую потерю производительности, а использование `x0` в качестве регистра назначения — плохой стиль программирования.

Разрешение конфликтов управления

Выполнение команды `beq` приводит к конфликту управления: конвейерный процессор не знает, какую команду выбрать следующей, поскольку в этот момент еще не ясно, нужно будет выполнить условный переход или нет.

Один из способов разрешить этот конфликт – приостановить конвейер до тех пор, пока не будет принято нужное решение (т. е. до тех пор, пока не будет вычислен сигнал *PCSrcE*). Поскольку решение принимается на стадии *Execute*, конвейер должен останавливаться на два такта для каждой команды условного перехода. Такое решение может серьезно снизить производительность системы, если ветвления будут происходить часто, что встречается нередко.

Есть и альтернативный способ – предсказать, будет выполнен условный переход или нет, и начать выполнять команды, основываясь на этом. Как только условие перехода будет вычислено, процессор может прервать эти команды, если предсказание было неверным. В конвейере, который мы рассматривали до сих пор (**рис. 7.58**), процессор предсказывает, что переходы не выполняются, и просто продолжает выполнять команды в порядке следования до тех пор, пока сигнал *PCSrcE* не укажет ему выбрать адрес перехода из *PCTargetE*. Если окажется, что переход должен был быть выполнен, то конвейер должен быть очищен (flushed) от трех команд, идущих сразу за командой перехода, путем очистки соответствующих временных регистров конвейера. Зря потраченные в этом случае такты называются простоем, или штрафом из-за неправильно предсказанного перехода (branch misprediction penalty).

На **рис. 7.59** показано, что происходит в конвейере, если выполнен условный переход из адреса 0x20 к адресу 0x58. Содержимое PC не меняется до третьего такта, к которому уже были извлечены команды `sub` и `or` по адресам 0x24 и 0x28 соответственно. Конвейер должен быть очищен от этих команд, после чего на четвертом такте будет выбрана команда `add` по адресу 0x58.

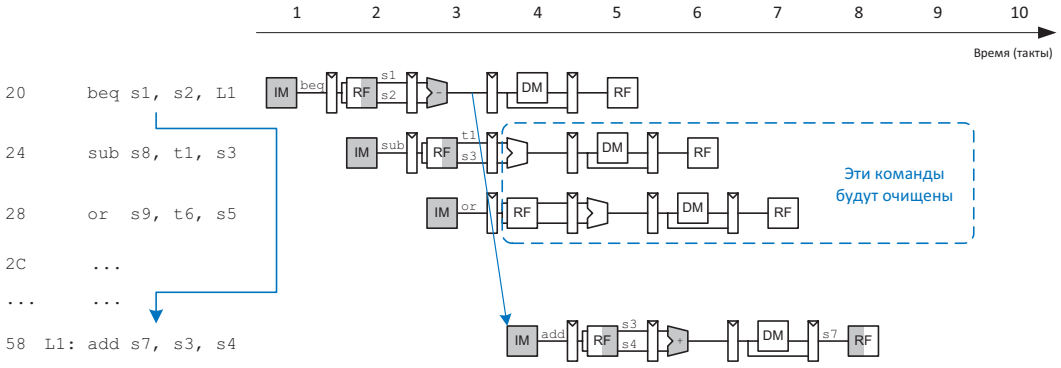


Рис. 7.59 Очистка конвейера при выполненном условном переходе

Наконец, мы должны разработать логику формирования сигналов приостановки и очистки для обработки переходов и записи нового адреса в счетчик команд. Когда выполняется переход, регистры конвейера должны быть очищены от следующих двух команд на стадиях *Decode* и *Execute*. Поэтому мы добавляем синхронный вход очистки (*CLR*) в регистр конвейера *Decode* и добавляем выход *FlushD* в блок разрешения конфликтов. (Когда *CLR* = 1, содержимое регистра обнуляется.) При переходе (когда *PCSrcE* = 1) должны быть установлены сигналы *FlushD* и *FlushE*, что приводит к очистке регистров конвейера *Decode* и *Execute*. На **рис. 7.60** показан модифицированный конвейерный процессор для обработки конфликтов. Логика формирования сигнала очистки теперь выглядит так:

$$FlushD = PCSrcE$$
$$FlushE = lwStall \mid PCSrcE$$

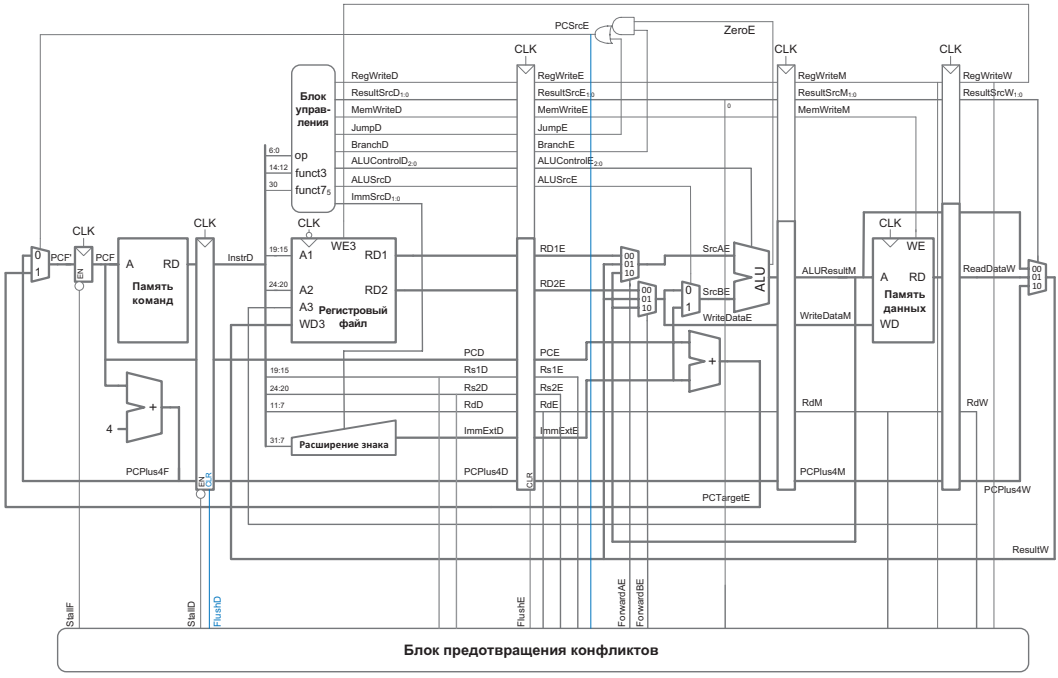


Рис. 7.60 Модифицированный конвейерный процессор с обработкой конфликтов при переходах

Промежуточные итоги

Конфликты при чтении данных после записи (RAW data hazards) случаются, когда одна из команд зависит от результата предыдущей команды, еще не записанного в регистровый файл. Такие конфликты можно разрешить при помощи байпаса, если результат вычислен достаточно рано; в противном случае требуется приостановка конвейера до тех пор, пока он

не будет готов. Конфликты управления возникают, когда нужно выбрать из памяти команду, а решение о том, какую именно, еще не принято. Эти конфликты разрешаются путем предсказания того, какая именно команда должна быть выбрана, и очисткой конвейера от ошибочно выбранных команд в случае, если предсказание не сбылось. Количество команд, от которых придется очистить конвейер в случае неправильно предсказанного перехода, минимизируется путем перемещения логики вычисления условия переходов в начало конвейера. Как вы могли заметить, при разработке конвейерных процессоров нужно понимать, как различные команды могут взаимодействовать между собой, а также заранее обнаруживать все возможные конфликты. На **рис. 7.61** показан конвейерный процессор, способный разрешить все имеющиеся конфликты.

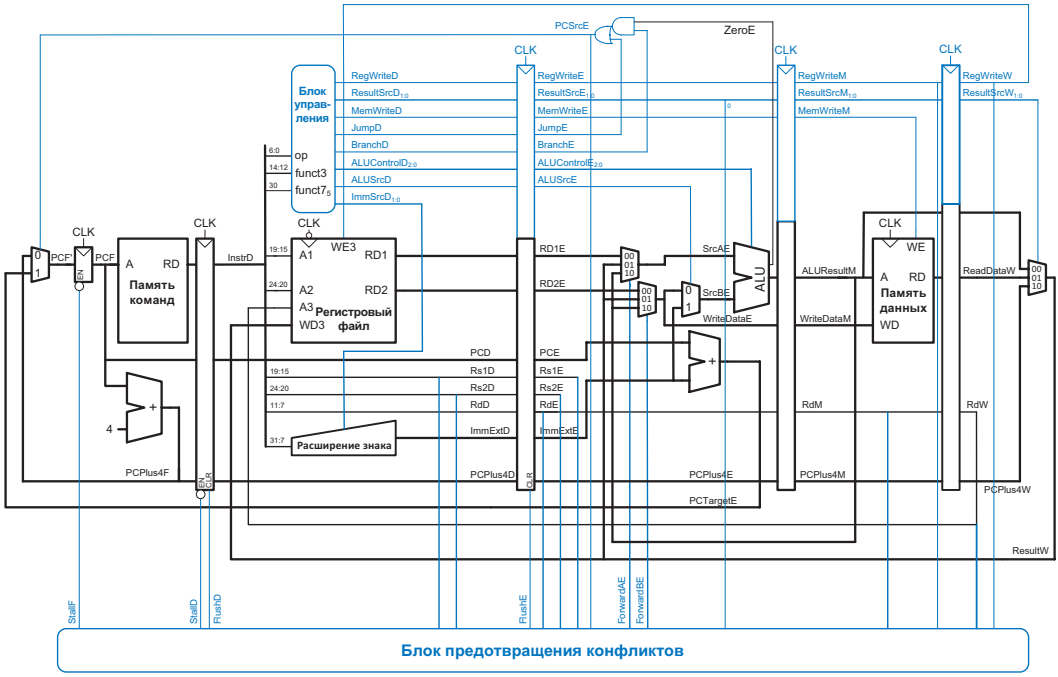


Рис. 7.61 Конвейерный процессор, разрешающий все конфликты

Полная логика формирования управляющих сигналов выглядит следующим образом.

Пересылка для разрешения конфликта данных, если это возможно[1]**:**

if $((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E != 0)$ then
$ForwardAE = 10$

[1] Напомним, что логика пересылки для *SrcBE* (*ForwardBE*) идентична, за исключением того, что она проверяет *Rs2E* вместо *Rs1E*.

else if ((*Rs1E* = = *RdW*) & *RegWriteW*) & (*Rs1E* != 0) then
ForwardAE = 01
else *ForwardAE* = 00

Приостановка в ожидании команды чтения из памяти:
lwStall = *ResultSrcE0* & ((*Rs1D* = = *RdE*) | (*Rs2D* = = *RdE*))
StallF = *lwStall*
StallD = *lwStall*

Очистка, если выполняется ветвление или возник пузырек при выполнении команды чтения из памяти:
FlushD = *PCSrcE*
FlushE = *lwStall* | *PCSrcE*

7.5.4. Анализ производительности

В идеальном случае у конвейерного процессора количество тактов на команду должно быть равно единице (CPI = 1), потому что на каждом такте процессор начинает выполнять новую команду. Но приостановки и очистки конвейера приводят к тому, что некоторые такты пропадают, так что в реальной жизни CPI немного больше, при этом она зависит от выполняемой программы.

Пример 7.9 CPI КОНВЕЙЕРНОГО ПРОЦЕССОРА

Бенчмарк SPECINT2000, рассмотренный в **примере 7.4**, содержит примерно 25 % команд чтения из памяти, 10 % команд записи в память, 11 % команд условного перехода, 2 % команд безусловного перехода и 52 % команд типа R. Предположим, что в 40 % случаев результат команды чтения из памяти требуется непосредственно следующей за ней команде, что приводит к приостановке конвейера. Также предположим, что 50 % всех условных переходов предсказываются неверно, что приводит к очистке конвейера. Прочими конфликтами пренебрежем. Требуется вычислить среднее количество тактов на команду (cycles per instruction, CPI) в конвейерном процессоре.

Решение Среднее CPI можно вычислить как взвешенную сумму числа тактов каждой команды, умноженного на долю, которую занимает эта команда в наборе команд программы. Команды чтения из памяти данных выполняются за один такт, если нет конфликтов, и за два такта, если конвейер должен быть приостановлен для разрешения конфликта, так что их CPI (0,6)(1) + (0,4)(2) = 1,4. Команды условного перехода выполняются за один такт, если переход предсказан корректно, и за два такта в противном случае, так что их CPI (0,5)(1) + (0,5)(3) = 2. Команды безусловного перехода занимают три такта (CPI = 3). Для всех остальных команд CPI = 1. Следовательно, для этого бенчмарка средний CPI = (0,25)(1,4) + (0,1)(1) + (0,11)(2) + (0,02)(3) + (0,52)(1) = 1,25.

Минимальную длительность такта мы можем определить, оценив наиболее длинную цепь в каждой из пяти стадий, показанных на **рис. 7.61**. Напомним, что процессор обращается к регистровому файлу дважды за

Анализ критического пути на стадии *Execute* предполагает, что задержка блока предотвращения конфликтов для выработки *ForwardAE* и *ForwardBE* меньше или равна задержке мультиплексора Result. Если, напротив, задержка блока предотвращения конфликтов больше, то она должна быть включена в критический путь вместо задержки мультиплексора Result.

один такт, выполняя запись на стадии *Writeback* в первой половине такта и чтение на стадии *Decode* во второй половине такта; таким образом, эти стадии могут использовать только половину длительности такта для прохождения критического пути. Иными словами, минимальная длительность такта в стадиях *Decode* и *Writeback* равна удвоенному времени чтения или записи соответственно, включая все накладные расходы на мультиплексирование данных и тому подобное. На **рис. 7.62** показан критический путь для стадии *Execute*. Он отражает ситуацию, когда условный переход находится на стадии *Execute*, требующей пересылки со стадии *Writeback*: путь идет от конвейерного регистра *Writeback* через мультиплексоры *Result*, *ForwardBE* и *SrcB*, через АЛУ и логику И-ИЛИ к мультиплексору PC и, наконец, к регистру PC.

$$T_{c\_pipelined} = \max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & Fetch \\ 2(t_{RFread} + t_{setup}) & Decode \\ t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup} & Execute \\ t_{pcq} + t_{mem} + t_{setup} & Memory \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & Writeback \end{bmatrix}. \quad (7.5)$$

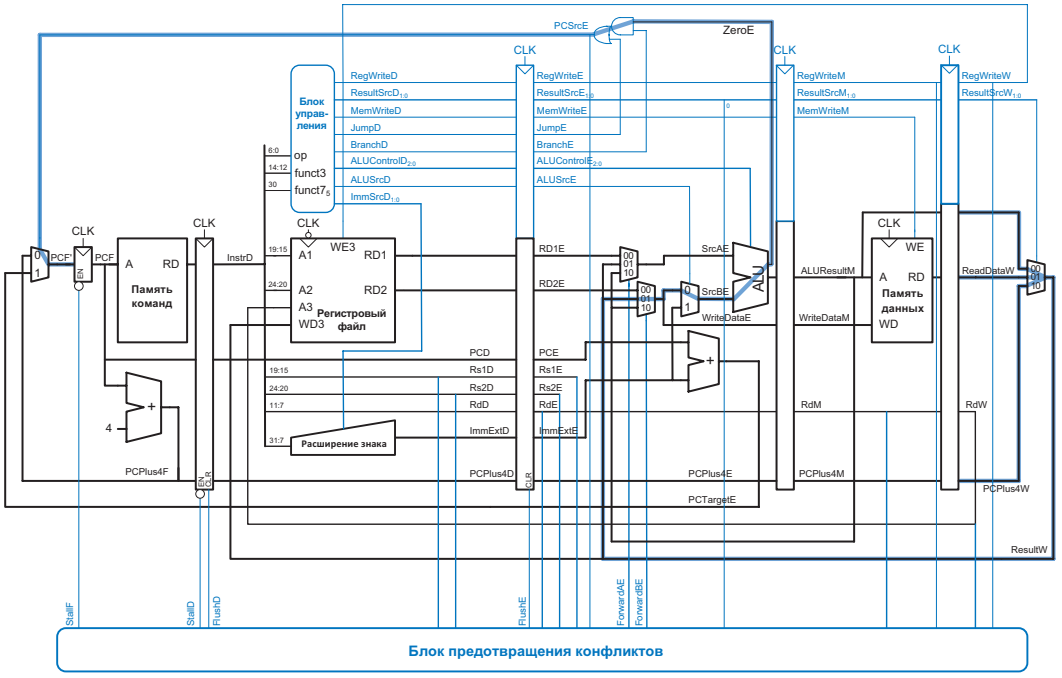


Рис. 7.62 Критический путь конвейерного процессора

Пример 7.10 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл хочет сравнить производительность конвейерного процессора с производительностью однотактного и многотактного процессоров, рассмотренных в **примерах 7.4** и **7.8**. Логические задержки приведены в **табл. 7.7**. Помогите Бену сравнить время выполнения 100 млрд команд из бенчмарка SPECINT2000 на каждом из этих процессоров.

Решение Согласно формуле **(7.5)**, длительность такта конвейерного процессора равна

$$T_{c\_pipelined} = \max[40 + 200 + 50, 2(100 + 50), 40 + 4(30) + 120 + 20 + 50, 40 + 200 + 50, 2(40 + 30 + 60)] = 350 \text{ пс}.$$

Стадия *Execute* занимает больше всего времени. Согласно формуле **(7.1)** общее время выполнения равно

$$T_{pipelines} = (100 \times 10^9 \text{ команд})(1{,}25 \text{ такта} / \text{команду})(350 \times 10^{-12} \text{ с} / \text{такт}) = 44 \text{ с}.$$

Время выполнения для однотактного процессора было равно 75 с, а для многотактного – 155 с.

Таким образом, конвейерный процессор значительно быстрее остальных. Но о пятикратном преимуществе над однотактным процессором, которое мы надеялись получить благодаря пятистадийному конвейеру, нет и речи.

Конфликты в конвейере вносят свою лепту в увеличение CPI; другая причина в том, что для организации конвейера нам потребовалось добавить временные (неархитектурные) регистры между стадиями, поэтому накладные расходы из-за использования регистров, равные сумме времени предустановки и времени задержки (clk-to-Q), теперь добавляются к каждой стадии, а не к общему времени выполнения команды. То, что больший процент длительности такта уходит на накладные расходы, ограничивает преимущества конвейерной обработки. Основные компоненты конвейерного процессора те же, что и у однотактного, но ему вдобавок требуется значительное количество временных регистров между стадиями, мультиплексоров и логики разрешения конфликтов.

Наш конвейерный процессор несбалансирован, и обработка условных переходов на стадии выполнения занимает гораздо больше времени, чем операции на других стадиях. Конвейер будет более сбалансированным, если мы вернем мультиплексор Result обратно в стадию *Memory*, сократив время цикла до 320 пс.

7.6. Разрабатываем процессор на HDL

В этом разделе приведен HDL-код однотактного процессора RISC-V, который поддерживает все рассмотренные в этой главе команды, включая `addi` и `j`. Этот код послужит примером хорошего стиля описания си-

стем умеренной сложности. Разработку HDL-кода для многотактного и конвейерного процессоров мы оставим читателю в качестве **упражнений 7.25–7.27** и **7.42–7.44**.

В этом разделе мы будем считать, что и память команд, и память данных находятся снаружи процессора и подсоединены к нему с помощью шин адреса и данных. На практике большинство процессоров извлекают команды и данные из отдельных кешей. Но для поддержки карт памяти меньшего размера, где данные могут размещаться вместе с командами, универсальный процессор также должен иметь возможность считывать данные из памяти команд. В **главе 8** мы вернемся к изучению систем памяти, включая взаимодействие кешей с основной памятью.

На **рис. 7.63** показана диаграмма однотактного процессора RISC-V, подключенного к внешней памяти. Процессор состоит из тракта данных, показанного на **рис. 7.15**, и устройства управления, показанного на **рис. 7.16**. Устройство управления, в свою очередь, состоит из основного декодера и декодера АЛУ. HDL-код поделен на несколько частей, каждая из которых описана в отдельном разделе. **Раздел 7.6.1** содержит описание тракта данных и устройства управления. **Раздел 7.6.2** описывает универсальные строительные блоки, такие как регистры и мультиплексоры, которые используются для любой микроархитектуры. **Раздел 7.6.3** демонстрирует код тестбенча и внешней памяти. HDL-код доступен в виде файлов для скачивания на веб-сайте книги (описано в **предисловии**).

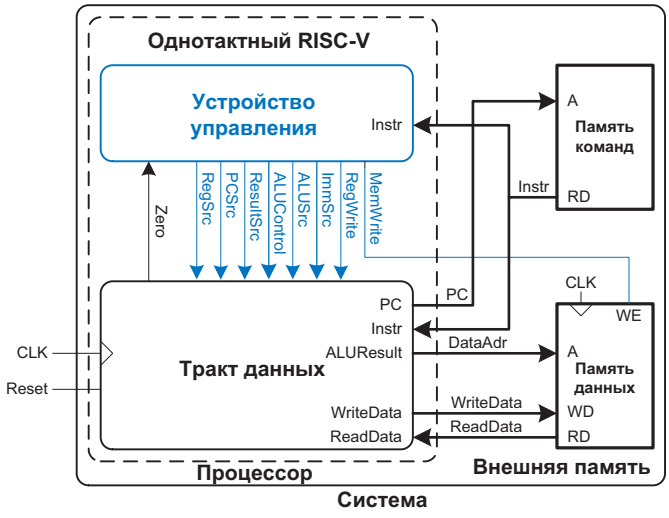


Рис. 7.63 Однотактный процессор, подключенный к внешней памяти

7.6.1. Однотактный процессор

Основные модули однотактного процессора RISC-V приведены ниже.

HDL-пример 7.1 ОДНОТАКТНЫЙ ПРОЦЕССОР RISC-V

SystemVerilog

```
module riscvsingle(input logic clk, reset,
                   output logic [31:0] PC,
                   input logic [31:0] Instr,
                   output logic MemWrite,
                   output logic [31:0]
                   ALUResult, WriteData,
                   input logic [31:0]
ReadData);

  logic       ALUSrc, RegWrite, Jump, Zero;
  logic [1:0] ResultSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(Instr[6:0], Instr[14:12],
               Instr[30], Zero,
               ResultSrc, MemWrite, PCSrc,
               ALUSrc, RegWrite, Jump,
               ImmSrc, ALUControl);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrc, RegWrite,
              ImmSrc, ALUControl,
              Zero, PC, Instr,
              ALUResult, WriteData,
ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity riscvsingle is
  port(clk, reset:           in STD_LOGIC;
       PC:                   out STD_LOGIC_VECTOR(31 downto 0);
       Instr:                in STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:             out STD_LOGIC;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:             in STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of riscvsingle is
  component controller
    port(op:             in STD_LOGIC_VECTOR(6 downto 0);
         funct3:         in STD_LOGIC_VECTOR(2 downto 0);
         funct7b5, Zero: in STD_LOGIC;
         ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:       out STD_LOGIC;
         PCSrc, ALUSrc:  out STD_LOGIC;
         RegWrite, Jump: out STD_LOGIC;
         ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:     out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset:           in STD_LOGIC;
         ResultSrc:            in STD_LOGIC_VECTOR(1 downto 0);
         PCSrc, ALUSrc:        in STD_LOGIC;
         RegWrite:             in STD_LOGIC;
         ImmSrc:               in STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:           in STD_LOGIC_VECTOR(2 downto 0);
         Zero:                 out STD_LOGIC;
         PC:                   out STD_LOGIC_VECTOR(31 downto 0);
         Instr:                in STD_LOGIC_VECTOR(31 downto 0);
         ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
         ReadData:             in STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal ALUSrc, RegWrite, Jump, Zero, PCSrc: STD_LOGIC;
  signal ResultSrc, ImmSrc: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUControl: STD_LOGIC_VECTOR(2 downto 0);
begin
  c: controller port map(Instr(6 downto 0), Instr(14 downto 12),
                         Instr(30), Zero, ResultSrc, MemWrite,
                         PCSrc, ALUSrc, RegWrite, Jump,
                         ImmSrc, ALUControl);
  dp:  datapath port map(clk, reset, ResultSrc, PCSrc, ALUSrc,
                         RegWrite, ImmSrc, ALUControl, Zero,
                         PC, Instr, ALUResult, WriteData,
                         ReadData);
end;
```

HDL-пример 7.2 УСТРОЙСТВО УПРАВЛЕНИЯ

SystemVerilog

```
module controller(input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic funct7b5,
                  input logic Zero,
                  output logic [1:0] ResultSrc,
                  output logic MemWrite,
                  output logic PCSrc, ALUSrc,
                  output logic RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl);
  logic [1:0] ALUOp;
  logic Branch;

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrc, RegWrite, Jump, ImmSrc,
             ALUOp);
  aludec ad(op[5], funct3, funct7b5, ALUOp,
            ALUControl);

  assign PCSrc = Branch & Zero | Jump;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity controller is
  port(op:             in STD_LOGIC_VECTOR(6 downto 0);
       funct3:         in STD_LOGIC_VECTOR(2 downto 0);
       funct7b5, Zero: in STD_LOGIC;
       ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       PCSrc, ALUSrc:  out STD_LOGIC;
       RegWrite:       out STD_LOGIC;
       Jump:           buffer STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:     out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(op:             in STD_LOGIC_VECTOR(6 downto 0);
         ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:       out STD_LOGIC;
         Branch, ALUSrc: out STD_LOGIC;
         RegWrite, Jump: out STD_LOGIC;
         ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
         ALUOp:          out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component aludec
    port(opb5:       in STD_LOGIC;
         funct3:     in STD_LOGIC_VECTOR(2 downto 0);
         funct7b5:   in STD_LOGIC;
         ALUOp:      in STD_LOGIC_VECTOR(1 downto 0);
         ALUControl: out STD_LOGIC_VECTOR(2 downto 0));
  end component;

  signal ALUOp:  STD_LOGIC_VECTOR(1 downto 0);
  signal Branch: STD_LOGIC;
begin
  md: maindec port map(op, ResultSrc, MemWrite, Branch,
                       ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  ad: aludec port map(op(5), funct3, funct7b5, ALUOp, ALUControl);
  PCSrc <= (Branch and Zero) or Jump;
end;
```

HDL-пример 7.3 ОСНОВНОЙ ДЕКОДЕР

SystemVerilog

```
module maindec(input logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic MemWrite,
               output logic Branch, ALUSrc,
               output logic RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity maindec is
  port(op:             in STD_LOGIC_VECTOR(6 downto 0);
       ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
```

HDL-пример 7.3 (окончание)

```
  logic [10:0] controls;
  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;
  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_
Branch_ALUOp_Jump
       7'b0000011: controls =
         11'b1_00_1_0_01_0_00_0; // lw
       7'b0100011: controls =
         11'b0_01_1_1_00_0_00_0; // sw
       7'b0110011: controls =
         11'b1_xx_0_0_00_0_10_0; // тип R
       7'b1100011: controls =
         11'b0_10_0_0_00_1_01_0; // beq
       7'b0010011: controls =
         11'b1_00_1_0_00_0_10_0; // тип I
       7'b1101111: controls =
         11'b1_11_0_0_10_0_00_1; // jal
       default: controls =
         11'bx_xx_x_x_xx_x_xx_x; // ???
    endcase
endmodule
```

```
        Branch, ALUSrc: out STD_LOGIC;
        RegWrite, Jump: out STD_LOGIC;
        ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
        ALUOp:          out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(10 downto 0);
begin
  process(op) begin
    case op is
      when "0000011" => controls <= "10010010000"; -- lw
      when "0100011" => controls <= "00111000000"; -- sw
      when "0110011" => controls <= "1--00000100"; -- тип R
      when "1100011" => controls <= "01000001010"; -- beq
      when "0010011" => controls <= "10010000100"; -- тип I
      when "1101111" => controls <= "11100100001"; -- jal
      when others => controls <= "-----------"; -- not valid
    end case;
  end process;

  (RegWrite, ImmSrc(1), ImmSrc(0), ALUSrc, MemWrite,
  ResultSrc(1), ResultSrc(0), Branch, ALUOp(1), ALUOp(0),
  Jump) <= controls;
end;
```

HDL-пример 7.4 ДЕКОДЕР АЛУ

SystemVerilog

```
module aludec(input logic opb5,
              input logic [2:0] funct3,
              input logic funct7b5,
              input logic [1:0] ALUOp,
              output logic [2:0] ALUControl);
  logic RtypeSub;
  assign RtypeSub = funct7b5 & opb5;
       // ИСТИНА для операции вычитания типа R

  always_comb
    case(ALUOp)
      2'b00:   ALUControl = 3'b000;
                            // сложение
      2'b01:   ALUControl = 3'b001;
                            // вычитание
      default: case(funct3) // тип R или I
               3'b000: if (RtypeSub)
                         ALUControl = 3'b001;
                            // sub
                       else
                         ALUControl = 3'b000;
                            // add, addi
               3'b010:  ALUControl = 3'b101;
                            // slt, slti
               3'b110:  ALUControl = 3'b011;
                            // or, ori
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity aludec is
  port(opb5:       in STD_LOGIC;
       funct3:     in STD_LOGIC_VECTOR(2 downto 0);
       funct7b5:   in STD_LOGIC;
       ALUOp:      in STD_LOGIC_VECTOR(1 downto 0);
       ALUControl: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture behave of aludec is
  signal RtypeSub: STD_LOGIC;
begin
  RtypeSub <= funct7b5 and opb5; -- ИСТИНА для операции вычитания типа R
  process(opb5, funct3, funct7b5, ALUOp, RtypeSub) begin
    case ALUOp is
    when "00" => ALUControl <= "000"; -- сложение
    when "01" => ALUControl <= "001"; -- вычитание
    when others => case funct3 is -- тип I или R
          when "000" = if RtypeSub = '1' then
                          ALUControl <= "001"; -- sub
                       else
                          ALUControl <= "000"; -- add, addi
                       end if;
          when "010" => ALUControl <= "101"; -- slt, slti
```

HDL-пример 7.4 (окончание)

```
            3'b111:  ALUControl = 3'b010;
                     // and, andi
            default: ALUControl = 3'bxxx;
                     // ???
          endcase
    endcase
endmodule
```

```
            when "110" => ALUControl <= "011"; -- or, ori
            when "111" => ALUControl <= "010"; -- and, andi
            when others => ALUControl <= "---"; -- неизвестно
          end case;
      end case;
    end process;
end;
```

HDL-пример 7.5 ТРАКТ ДАННЫХ

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc, ALUSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                output logic        Zero,
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult,
                                    WriteData,
                input  logic [31:0] ReadData);


  logic [31:0] PCNext, PCPlus4, PCTarget;
  logic [31:0] ImmExt;
  logic [31:0] SrcA, SrcB;
  logic [31:0] Result;

  // логика PC
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder       pcadd4(PC, 32'd4, PCPlus4);
  adder       pcaddbranch(PC, ImmExt, PCTarget);
  mux2 #(32)  pcmux(PCPlus4, PCTarget, PCSrc,
                    PCNext);

  // логика регистрового файла
  regfile     rf(clk, RegWrite, Instr[19:15],
                 Instr[24:20], Instr[11:7], Result,
                 SrcA, WriteData);
  extend      ext(Instr[31:7], ImmSrc, ImmExt);

  // логика АЛУ
  mux2 #(32)  srcbmux(WriteData, ImmExt, ALUSrc,
                      SrcB);
  alu         alu(SrcA, SrcB, ALUControl,
                  ALUResult, Zero);
  mux3 #(32)  resultmux(ALUResult, ReadData,
                        PCPlus4,
                        ResultSrc, Result);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity datapath is
  port(clk, reset:           in     STD_LOGIC;
       ResultSrc:            in     STD_LOGIC_VECTOR(1 downto 0);
       PCSrc, ALUSrc:        in     STD_LOGIC;
       RegWrite:             in     STD_LOGIC;
       ImmSrc:               in     STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:           in     STD_LOGIC_VECTOR(2 downto 0);
       Zero:                 out    STD_LOGIC;
       PC:                   buffer STD_LOGIC_VECTOR(31 downto 0);
       Instr:                in     STD_LOGIC_VECTOR(31 downto 0);
       ALUResult, WriteData: buffer STD_LOGIC_VECTOR(31 downto 0);
       ReadData:             in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux3 generic(width: integer);
    port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:          in  STD_LOGIC_VECTOR(1 downto 0);
         y:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component regfile
    port(clk: in STD_LOGIC;
         we3: in STD_LOGIC;
         a1, a2, a3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:        in  STD_LOGIC_VECTOR(31 downto 0);
```

HDL-пример 7.5 (окончание)

```
        rd1, rd2:   out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
    port(instr:  in  STD_LOGIC_VECTOR(31 downto 7);
         immsrc: in  STD_LOGIC_VECTOR(1 downto 0);
         immext: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component alu
    port(a, b:       in     STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in     STD_LOGIC_VECTOR(2 downto 0);
         ALUResult:  buffer STD_LOGIC_VECTOR(31 downto 0);
         Zero:       out    STD_LOGIC);
  end component;
 signal PCNext, PCPlus4, PCTarget: STD_LOGIC_VECTOR(31 downto 0);
  signal ImmExt: STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB: STD_LOGIC_VECTOR(31 downto 0);
  signal Result: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- логика PC
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd4: adder port map(PC, X"00000004", PCPlus4);
  pcaddbranch: adder port map(PC, ImmExt, PCTarget);
  pcmux: mux2 generic map(32) port map(PCPlus4, PCTarget, PCSrc,
                                       PCNext);
  -- логика регистрового файла
  rf: regfile port map(clk, RegWrite, Instr(19 downto 15),
                       Instr(24 downto 20), Instr(11 downto 7),
                       Result, SrcA, WriteData);
  ext: extend port map(Instr(31 downto 7), ImmSrc, ImmExt);
  -- ALU logic
  srcbmux: mux2 generic map(32) port map(WriteData, ImmExt,
                                         ALUSrc, SrcB);
  mainalu: alu port map(SrcA, SrcB, ALUControl, ALUResult, Zero);
  resultmux: mux3 generic map(32) port map(ALUResult, ReadData,
                                           PCPlus4, ResultSrc,
                                           Result);
end;
```

7.6.2. Универсальные строительные блоки

Этот раздел содержит универсальные строительные блоки, которые могут быть полезны для реализации любой микроархитектуры. Эти блоки включают сумматор, триггеры и мультиплексор 2:1. Регистровый файл рассмотрен в **HDL-примере 5.8**. Разработку HDL-кода для АЛУ мы оставили читателю в качестве **упражнений 5.11–5.14**.

HDL-пример 7.6 СУММАТОР

SystemVerilog

```
module adder(input [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity adder is
  port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL-пример 7.7 БЛОК РАСШИРЕНИЯ ЗНАКА

SystemVerilog

```
module extend(input logic [31:7] instr,
              input logic [1:0] immsrc,
              output logic [31:0] immext);
  always_comb
    case(immsrc)
               // тип I
      2'b00: immext = {{20{instr[31]}},
                       instr[31:20]};
               // тип S (запись в память)
      2'b01: immext = {{20{instr[31]}},
                       instr[31:25],
                       instr[11:7]};
               // тип B (условный переход)
      2'b10: immext = {{20{instr[31]}},
                       instr[7],
                       instr[30:25],
                       instr[11:8], 1'b0};
               // тип J (jal)
      2'b11: immext = {{12{instr[31]}},
                       instr[19:12],
                       instr[20],
                       instr[30:21], 1'b0};
      default: immext = 32'bx;
               // не определено
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity extend is
  port(instr: in STD_LOGIC_VECTOR(31 downto 7);
       immsrc: in STD_LOGIC_VECTOR(1 downto 0);
       immext: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(instr, immsrc) begin
    case immsrc is
      -- тип I
      when "00" =>
        immext <= (31 downto 12 => instr(31)) & instr(31 downto 20);
      -- тип S (запись в память)
      when "01" =>
         immext <= (31 downto 12 => instr(31)) &
                    instr(31 downto 25) & instr(11 downto 7);
      -- тип B (переходы)
      when "10" =>
        immext <= (31 downto 12 => instr(31)) & instr(7) & instr(30
                   downto 25) & instr(11 downto 8) & '0';
      -- тип J (jal)
      when "11" =>
        immext <= (31 downto 20 => instr(31)) &
                  instr(19 downto 12) & instr(20) &
                  instr(30 downto 21) & '0';
      when others =>
        immext <= (31 downto 0 => '-');
    end case;
  end process;
end;
```

HDL-пример 7.8 ТРИГГЕР С СИГНАЛОМ СБРОСА

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

   always_ff @(posedge clk, posedge reset)
               if (reset) q <= 0;
               else q <= d;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopr is
  generic(width: integer);
  port(clk, reset: in STD_LOGIC;
       d:          in STD_LOGIC_VECTOR(width-1
                                       downto 0);
       q:          out STD_LOGIC_VECTOR(width-1
                                        downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then q <= (others => '0');
    elsif rising_edge(clk) then q <= d;
    end if;
  end process;
end;
```

HDL-пример 7.9 ТРИГГЕР С СИГНАЛОМ СБРОСА И ВХОДОМ ВЫБОРА

SystemVerilog

```
module flopenr #(parameter WIDTH = 8)
                (input logic clk, reset, en,
                 input logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);
always_ff @(posedge clk, posedge reset)
  if (reset) q <= 0;
  else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopenr is
  generic(width: integer);
  port(clk, reset, en: in STD_LOGIC;
       d:              in STD_LOGIC_VECTOR(width-1 downto 0);
       q:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset, en) begin
    if reset = '1' then q <= (others => '0');
    elsif rising_edge(clk) and en = '1' then q <= d;
    end if;
  end process;
end;
```

HDL-пример 7.10 МУЛЬТИПЛЕКСОР 2:1

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d0, d1,
              input logic s,
              output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s = '1' else d0;
end;
```

HDL-пример 7.11 МУЛЬТИПЛЕКСОР 3:1

SystemVerilog

```
module mux3 #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d0, d1, d2,
              input logic [1:0] s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux3 is
  generic(width: integer := 8);
  port(d0, d1, d2: in STD_LOGIC_VECTOR(width-1 downto 0);
       s:          in STD_LOGIC_VECTOR(1 downto 0);
       y:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux3 is
begin
  process(d0, d1, d2, s) begin
    if (s = "00") then y <= d0;
    elsif (s = "01") then y <= d1;
    elsif (s = "10") then y <= d2;
    end if;
  end process;
end;
```

7.6.3. Тестбенч

Тестбенч загружает в память команд программу, которая использует все команды процессора и выполняет вычисления, приводящие к корректному результату только в том случае, когда процессор работает правильно. В случае успешного выполнения программа запишет значение 25 в адрес 100, но вряд ли сделает это, если процессор содержит ошибки. Это – пример *направленного тестирования* (ad hoc testing).

```
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020
#
# Тестбенч процессора RISC-V:
# add, sub, and, or, slt, addi, lw, sw, beq, jal
# Если выполнен успешно, то значение 25 будет записано по адресу 100
#         Ассемблер RISC-V   Описание                    Адрес      Маш. код
main:     addi x2, x0,5      # x2 = 5                     0          00500113
          addi x3, x0, 12    # x3 = 12                    4          00C00193
          addi x7, x3, -9    # x7 = (12 - 9) = 3          8          FF718393
          or x4, x7, x2      # x4 = (3 OR 5) = 7          C          0023E233
          and x5, x3, x4     # x5 = (12 AND 7) = 4        10         0041F2B3
          add x5, x5, x4     # x5 = 4 + 7 = 11            14         004282B3
          beq x5, x7, end    # не должно случиться        18         02728863
          slt x4, x3, x4     # x4 = (12 < 7) = 0          1C         0041A233
          beq x4, x0, around # должно случиться          20         00020463
          addi x5, x0,0      # не выполняется             24         00000293
around:   slt x4, x7, x2     # x4 = (3 < 5) = 1           28         0023A233
          add x7, x4, x5     # x7 = (1 + 11) = 12         2C         005203B3
          sub x7, x7, x2     # x7 = (12 - 5) = 7          30         402383B3
          sw x7, 84(x3)      # [96] = 7                   34         0471AA23
          lw x2, 96(x0)      # x2 = [96] = 7              38         06002103
          add x9, x2, x5     # x9 = (7 + 11) = 18         3C         005104B3
          jal x3, end        # переход к end, x3 = 0x44   40         008001EF
          addi x2, x0,1      # не выполняется             44         00100113
end:      add x2, x2, x9     # x2 = (7 + 18) = 25         48         00910133
          sw x2, 0x20(x3)    # [100] = 25                 4C         0221A023
done:     beq x2, x2, done   # бесконечный цикл           50         00210063
```

Рис. 7.64 Ассемблерный и машинный коды тестовой программы riscvtest.s

Машинный код хранится в шестнадцатеричном файле `riscvtest.txt` (**рис. 7.65**), который загружается программой тестбенча. Файл состоит из машинного кода команд процессора, в каждой строке представлено по одной команде.

HDL-коды тестбенча, модуля верхнего уровня иерархии процессора RISC-V (который создает экземпляр (инстанс) процессора и памяти RISC-V), а также блоков внешней памяти приведены в следующих примерах. Тестбенч создает экземпляр проверяемого модуля верхнего уровня иерархии и генерирует тактовые импульсы и сигнал сброса в начале моделирования. Он проверяет наличие записи в память и сообщает об успехе, если правильное значение (25) записано по адресу 100. Память в этом примере содержит 64 слова по 32 бита каждое.

```
00500113
00C00193
FF718393
0023E233
0041F2B3
004282B3
02728863
0041A233
00020463
00000293
0023A233
005203B3
402383B3
0471AA23
06002103
005104B3
008001EF
00100113
00910133
0221A023
00210063
```

Рис. 7.65 Файл riscvtest.txt

HDL-пример 7.12 ТЕСТБЕНЧ ПРОЦЕССОРА RISC-V

SystemVerilog

```
module testbench();

  logic clk;
  logic reset;
  logic [31:0] WriteData, DataAdr;
  logic MemWrite;

  // инициализация проверяемого устройства
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // запуск тестбенча
  initial
    begin
      reset <= 1; # 22; reset <= 0;
  end

  // генерация тактовых импульсов
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
  end

  // проверка результата
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr = = = 100 & WriteData
           = = = 25) begin
          $display("Проверка успешно пройдена");
          $stop;
        end else if (DataAdr != = 96) begin
          $display("Обнаружена ошибка");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset: in STD_LOGIC;
         WriteData, DataAdr: out STD_LOGIC_VECTOR(31 downto 0);
         MemWrite: out STD_LOGIC);
  end component;

  signal WriteData, DataAdr: STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, MemWrite: STD_LOGIC;
begin
  -- создание образа проверяемого устройства
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- генерация тактовых импульсов с периодом 10 нс
  process begin
     clk <= '1';
     wait for 5 ns;
     clk <= '0';
     wait for 5 ns;
  end process;

  -- генерация сигнала сброса на протяжении первых двух тактов
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- проверка наличия значения 25 по адресу 100 после завершения
  process(clk) begin
    if(clk'event and clk = '0' and MemWrite = '1') then
      if(to_integer(DataAdr) = 100 and
         to_integer(writedata) = 25) then
        report "Ошибок не обнаружено" severity
        failure;
      elsif (DataAdr /= 96) then
        report "Обнаружена ошибка" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL-пример 7.13 МОДУЛЬ ВЕРХНЕГО УРОВНЯ ИЕРАРХИИ ПРОЦЕССОРА RISC-V

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData,
                               DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // инстанциирование процессора и памяти
  riscvsingle rvsingle(clk, reset, PC, Instr,
                       MemWrite, DataAdr,
                       WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData,
            ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is
  port(clk, reset:         in STD_LOGIC;
       WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:           buffer STD_LOGIC);
end;

architecture test of top is
  component riscvsingle
    port(clk, reset:        in STD_LOGIC;
      PC:                   out STD_LOGIC_VECTOR(31 downto 0);
      Instr:                in STD_LOGIC_VECTOR(31 downto 0);
      MemWrite:             out STD_LOGIC;
      ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:             in STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
  port(clk, we: in STD_LOGIC;
    a, wd:      in STD_LOGIC_VECTOR(31 downto 0);
    rd:         out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  signal PC, Instr, ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- инстанциирование процессора и памяти
  rvsingle: riscvsingle port map(clk, reset, PC, Instr,
                                 MemWrite, DataAdr,
                                 WriteData, ReadData);
  imem1: imem port map(PC, Instr);
  dmem1: dmem port map(clk, MemWrite, DataAdr, WriteData,
                       ReadData);
end;
```

HDL-пример 7.14 ПАМЯТЬ КОМАНД

SystemVerilog

```
module imem(input logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
use ieee.std_logic_textio.all;

entity imem is
  port(a: in STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;
```

HDL-пример 7.14 (окончание)

```
architecture behave of imem is
type ramtype is array (63 downto 0) of
                          STD_LOGIC_VECTOR(31 downto 0);
  -- инициализировать память из файла
  impure function init_ram_hex return ramtype is
  file text_file : text open read_mode is "riscvtest.txt";
    variable text_line : line;
    variable ram_content : ramtype;
    variable i : integer := 0;
  begin
    for i in 0 to 63 loop -- установить все значения в 0
      ram_content(i) := (others => '0');
    end loop;
    while not endfile(text_file) loop -- прочитать данные из файла
      readline(text_file, text_line);
      hread(text_line, ram_content(i));
      i := i + 1;
    end loop;


    return ram_content;
  end function;

  signal mem : ramtype := init_ram_hex;
  begin
  -- чтение памяти
  process(a) begin
    rd <= mem(to_integer(a(31 downto 2)));
  end process;
end;
```

HDL-пример 7.15 ПАМЯТЬ ДАННЫХ

SystemVerilog

```
module dmem(input  logic clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // выравнивание
                              // по слову

    always_ff @(posedge clk)
      if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                    STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- чтение или запись в память
    loop
```

HDL-пример 7.15 (окончание)

```
      if rising_edge(clk) then
        if (we = '1') then mem(to_integer(a(7 downto 2))) := wd;
        end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

7.7. Улучшенные микроархитектуры

Высокопроизводительные микропроцессоры используют большое разнообразие приемов, для того чтобы выполнять программы быстрее. Вспомним, что время, требуемое для выполнения программы, пропорционально периоду тактового сигнала, а также среднему количеству тактов на команду (clock cycles per instruction, CPI). Таким образом, чтобы увеличить производительность, необходимо либо ускорить тактовый сигнал, либо снизить CPI. Данный раздел представляет собой обзор некоторых способов достичь этого. Реализация данных способов является довольно сложной, поэтому в этом курсе мы сфокусируемся только на общих концепциях. Книга Хеннесси и Паттерсона «Архитектура компьютера и проектирование компьютерных систем» (СПб.: Питер, 2012) является наиболее авторитетным источником для тех, кто захочет полностью изучить все детали.

Достижения в технологии производства интегральных схем неуклонно ведут к сокращению размера транзисторов. С уменьшением размера транзисторы работают быстрее и, как правило, потребляют меньше электроэнергии. Таким образом, даже если не менять микроархитектуру, частота тактового сигнала может увеличиться просто потому, что все логические элементы стали быстрее. Кроме того, чем меньше по размеру транзисторы, тем больше их поместится на чипе. Разработчики микроархитектуры используют дополнительные транзисторы, чтобы строить более сложные процессоры или размещать больше процессоров в чипе. К сожалению, увеличение количества транзисторов и скорости, на которой они работают, вызывает увеличение энергопотребления. На настоящий момент энергопотребление стало одной из главных забот разработчиков микропроцессоров (**раздел 1.8**). Перед ними встает непростая задача добиться компромисса между скоростью, энергопотреблением и ценой микросхем, некоторые из которых содержат миллиарды транзисторов и являются одними из самых сложных систем, когда-либо созданных человечеством.

В конце 1990 — начале 2000-х годов благодаря усилиям маркетологов основным рыночным критерием при продаже микропроцессоров была тактовая частота ($f = 1/T_c$). Это привело к тому, что разработчики стали использовать очень длинные конвейеры (от 20 до 31 стадии на Pentium IV) для максимизации тактовой частоты, даже если выигрыш в общей производительности системы был сомнительным. Энергопотребление процессора пропорционально тактовой частоте и возрастает с увеличением числа регистров конвейера, поэтому теперь, когда потребление энергии стало более важным, длина конвейера современных процессоров стала меньше.

7.7.1. Длинные конвейеры

Помимо улучшения технологического процесса, простейший способ увеличить тактовую частоту процессора состоит в том, чтобы поделить конвейер на большее количество стадий. Каждая стадия в этом случае содержит меньше комбинационной логики и, следовательно, может работать быстрее. В начале этой главы мы рассмотрели классический пятистадийный конвейер, но в наши дни нередко можно увидеть конвейеры на 8–20 стадий. Например, ядро SweRV EH1 коммерческого процессора RISC-V с открытым исходным кодом, разработанного Western Digital, имеет девять стадий конвейера.

Максимальное количество стадий конвейера ограничено конфликтами в конвейере, накладными расходами на временные регистры (т. е. их временем предустановки и удержания), стоимостью разработки и производства. Более длинные конвейеры приводят к большему числу зависимостей внутри процессора. Некоторые из зависимостей могут быть разрешены при помощи байпаса (bypassing или forwarding), т. е. передачи результата из более поздних стадий конвейера в более ранние напрямую, минуя регистровый файл. Другие зависимости требуют приостановок (stalls), которые увеличивают CPI. Регистры, которые находятся между стадиями конвейера, приводят к накладным расходам из-за их времени предустановки (setup time) и задержки распространения (clk-to-Q delay), а также из-за сдвига фазы тактового сигнала (clock skew), вследствие чего добавление каждой последующей стадии дает все меньший прирост производительности. Наконец, добавление большего количества стадий увеличивает стоимость разработки и производства процессора, так как приходится добавлять регистры между стадиями и комбинационную логику для разрешения более сложных конфликтов.

Пример 7.11 ДЛИННЫЕ КОНВЕЙЕРЫ

Давайте построим конвейерный процессор, поделив однотактный процессор на N стадий. Задержка распространения сигнала через комбинационную логику однотактного процессора составляет 750 пс. Накладные расходы на использование регистров — 90 пс. Предположим, что комбинационная логика может быть поделена на произвольное количество стадий, а логика обнаружения конфликтов не увеличивает ее задержку. CPI пятистадийного конвейера из **примера 7.9** равен 1,25. Также предположим, что каждая дополнительная стадия увеличивает CPI на 0,1 из-за неправильного предсказания переходов и прочих конфликтов. При каком количестве стадий процессор будет выполнять программы быстрее всего?

Решение Длительность такта N-стадийного конвейера составляет $T_c = [(750/N) + 90]$ пс. CPI будет равно $1{,}25 + 0{,}1(N - 5)$, где $N \geq 5$. Время выполнения одной команды равно произведению длительности такта на CPI. На **рис. 7.66**

приведен график зависимости длительности такта и длительности выполнения команды (Instruction time) от количества стадий. Минимальная длительность выполнения команды равна 281 пс при количестве стадий $N = 8$. Это лишь незначительно лучше, чем 295 пс на команду, которых можно достичь, используя пятистадийный конвейер, и кривая графика зависимости почти плоская между 7-й и 10-й стадиями.

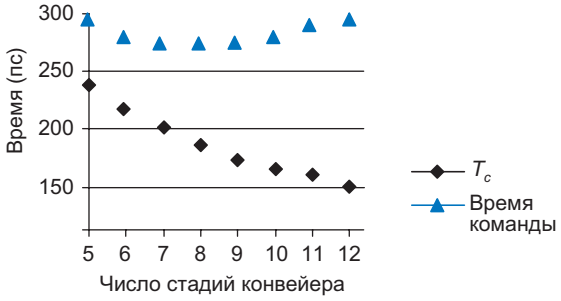


Рис. 7.66 Зависимость длительности такта и длительности выполнения команды от количества стадий

7.7.2. Микрокоманды

Вспомните наши правила разработки: «для простоты придерживайтесь единообразия» и «чем меньше, тем быстрее». Компьютерные архитектуры с *сокращенным набором команд* (reduced instruction set computer, RISC), к которым относится и RISC-V, содержат только простые инструкции, обычно те, которые могут быть выполнены за один цикл на простом и быстром тракте данных с трехпортовым регистровым файлом, одним АЛУ и одиночным обращением к памяти данных – наподобие процессора, который мы разработали в этой главе. Архитектура компьютеров с *полным набором команд* (complex instruction set computer CISC) обычно содержит инструкции, требующие большего количества регистров, большего количества операций сложения или более одного обращения к памяти для каждой команды. Например, выполнение инструкции процессора x86 `ADD [ESP], [EDX + 80 + EDI * 2]` включает чтение трех регистров (`ESP`, `EDX` и `EDI`), прибавление базы (`EDX`), смещения (80) и масштабированного индекса (`EDI*2`), чтение из памяти, сложение значений аргументов инструкции и запись результата обратно в память. Микропроцессор, способный параллелльно выполнять все эти действия, будет излишне медленным при выполнении более распространенных и простых инструкций.

Разработчики архитектуры процессоров CISC ускоряют выполнение распространенных операций, определяя

Разработчики процессорных микроархитектур постоянно ищут компромисс между прямой аппаратной реализацией сложной операции или дроблением этой операции на последовательность *микроопераций*. В результате появилось множество решений, обладающих разным соотношением между производительностью, энергопотреблением и стоимостью.

набор простых *микроопераций* (также известных как μops), которые используют упрощенный тракт данных. Каждая инструкция CISC декодируется в одну или несколько микроопераций. Например, если мы определим микрооперации, похожие на инструкции RISC-V, и воспользуемся временными регистрами `t1` и `t2` для хранения промежуточных результатов, то упомянутая выше инструкция x86 может превратиться в шесть микроопераций:

```
slli t2, EDI, 1  # t2 = EDI*2
add  t1, EDX, t2 # t1 = EDX + EDI*2
lw   t1, 80(t1)  # t1 = MEM[EDX + EDI*2 + 80]
lw   t2, 0(ESP)  # t2 = MEM[ESP]
add  t1, t2, t1  # t1 = MEM[ESP] + MEM[EDX + EDI*2 + 80]
sw   t1, 0(ESP)  # MEM[ESP] = MEM[ESP] + MEM[EDX + EDI*2 + 80]
```

7.7.3. Предсказание условных переходов

Теоретически CPI идеального конвейерного процессора должно быть равно единице. Одной из основных причин более высокого CPI являются простои процессора из-за неправильно предсказанных условных переходов (branch misprediction penalty). С увеличением длины конвейера необходимость перехода выясняется во все более поздних стадиях конвейера. Таким образом, простои из-за неправильно предсказанных переходов становятся все больше и больше, так как конвейер должен быть очищен (flushed) от всех команд, выбранных после неправильно предсказанного перехода. Чтобы разрешить эту проблему, большинство конвейерных процессоров используют предсказатель условных переходов, позволяющий с высокой вероятностью угадать, стоит ли осуществлять переход. Вспомним, что наш конвейер из **раздела 7.5.3** всегда считал, что переход осуществлять не стоит.

Некоторые условные переходы происходят тогда, когда программа доходит до конца цикла (т. е. в операторах `for` или `while`) и переходит к его началу для новой итерации. Циклы часто выполняются много раз, поэтому такие условные переходы назад, как правило, исполняются. Простейший метод предсказания условных переходов – это проверить направление перехода и считать, что переход назад всегда будет выполнен. Такой метод называется *статическим предсказанием переходов* (static branch prediction), потому что он не зависит от истории выполнения команд программы. Переходы вперед трудно предсказать без детального понимания конкретной программы. Из-за этого большинство процессоров используют *динамические предсказатели переходов* (dynamic branch predictors), которые применяют историю выполнения программы, для того чтобы предсказать, нужно ли выполнить переход. Динамические предсказатели переходов содержат таблицу с последними несколькими сотнями (или тысячами) команд условного перехода,

выполненными процессором. Эта таблица, которую иногда называют *буфером целевых адресов ветвлений* (branch target buffer), содержит адреса переходов и информацию о том, был ли переход выполнен.

Чтобы продемонстрировать работу динамического предсказателя переходов, рассмотрим код цикла из **примера 6.20**. Цикл повторяется 10 раз, причем выход из цикла (`bge s0, t0, done`) выполняется только на последней итерации.

```
      addi s1, zero, 0  # s1 = sum = 0
      addi s0, zero, 0  # s0 = i = 0
      addi t0, zero, 10 # t0 = 10
for:
      bge  s0, t0, done # i >= 10?
      add  s1, s1, s0   # sum = sum + i
      addi s0, s0, 1    # i = i + 1
      j    for          # повтор цикла
done:
```

Однобитный динамический предсказатель переходов (one-bit dynamic branch predictor) запоминает, был ли переход выполнен в прошлый раз, и предсказывает, что в следующий раз произойдет то же самое. Пока цикл повторяется, предсказатель помнит, что в прошлый раз команда beq не была выполнена, и предсказывает, что она не будет выполнена и в следующий раз. Это предсказание остается правильным вплоть до последней итерации, на которой переход все-таки выполняется. К сожалению, если цикл запустить снова, то предсказатель переходов будет помнить, что в последний раз условный переход был выполнен. Поэтому на первой итерации запущенного заново цикла предсказатель ошибется – неправильно предскажет, что переход нужно выполнить. Итого однобитный предсказатель переходов ошибется и на первой, и на последней итерациях этого цикла.

Двухбитный динамический предсказатель переходов решает эту проблему, используя четыре состояния: переход точно выполнится; переход, скорее, выполнится; переход, скорее, не выполнится и переход точно не выполнится (strongly taken, weakly taken, weakly not taken, strongly not taken), как показано на **рис. 7.67**. Пока цикл повторяется, предсказатель переходит в состояние «точно не выполнится» и остается в нем, предсказывая, что условный переход не будет выполнен и в следующий раз. Это предсказание остается верным вплоть до последней итерации, на которой условный переход выполняется и переводит предсказатель в состояние «скорее, не выполнится». Когда цикл начнется снова, предсказатель переходов правильно предскажет, что переход не должен быть выполнен, и снова перейдет в состояние «точно не выполнится». Таким образом, двухбитный предсказатель переходов неправильно предсказывает только переход на последней итерации цикла. Он называется двухбитным, потому что для кодирования четырех состояний требуется два бита.

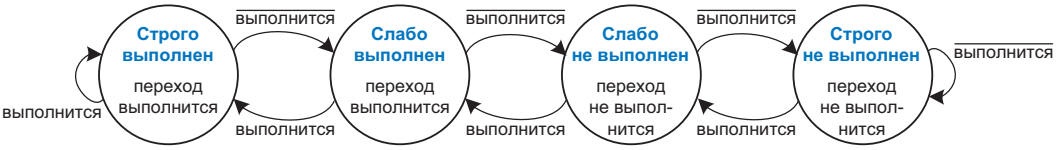


Рис. 7.67 Диаграмма состояний двухбитного предсказателя переходов

Скалярный процессор (scalar processor) в каждый момент времени осуществляет вычисления только над одной порцией данных. Векторный процессор (vector processor) работает над несколькими порциями данных одновременно, но использует для этого только одну команду. *Суперскалярный процессор* (superscalar processor) запускает на выполнение (issues) несколько команд одновременно, каждая из которых оперирует над одной порцией данных.

Скалярные процессоры относятся к категории машин с одним потоком команд и одним потоком данных (single-instruction single-data, SISD). Векторные процессоры и графические процессоры (graphics processing unit, GPU) представляют собой машины с одним потоком команд и несколькими потоками данных (single instruction multiple-data, SIMD). Мультипроцессоры, такие как многоядерные процессоры, относятся к машинам с несколькими потоками команд и данных (multiple instruction multiple-data, MIMD). Обычно на машинах MIMD работает одна программа, которая использует все или часть ядер. Этот стиль программирования называется парадигмой «одна программа, много данных» (single-program multiple-data, SPMD), но мультипроцессоры можно программировать и другими способами.

Предсказатель переходов работает на стадии выборки команд из памяти (стадия Fetch) конвейера и используется процессором, чтобы определить, какую команду выбирать на следующем такте. Когда предсказатель говорит, что условный переход будет выполнен, процессор выбирает следующую команду из ячейки памяти, адрес которой находится в *таблице адресов переходов* (branch target buffer).

Как легко представить, предсказатели переходов могут следить и за большей историей выполнения программы, тем самым повышая точность предсказаний. Для типичных программ точность хороших предсказателей переходов превышает 90 %.

7.7.4. Суперскалярный процессор

Тракт данных *суперскалярного процессора* содержит несколько копий функциональных блоков, что позволяет ему выполнять несколько команд одновременно. На **рис. 7.68** показана диаграмма двухконвейерного (2-way) суперскалярного процессора, который осуществляет выборку и выполнение двух команд за такт. Тракт данных выбирает из памяти команд две команды за один раз. Он содержит регистровый файл с шестью портами, чтобы читать четыре операнда и записывать назад два результата на каждом такте. Тракт данных также содержит два АЛУ и двухпортовую память данных, чтобы выполнять две команды одновременно.

На **рис. 7.69** приведена диаграмма двухканального суперскалярного процессора, который выполняет две инструкции на каждом такте. В этом случае CPI процессора равен 0,5. Разработчики зачастую используют величину, обратную CPI, – *количество команд на такт* (instructions per cycle, IPC), которое для этого процессора равно 2,0.

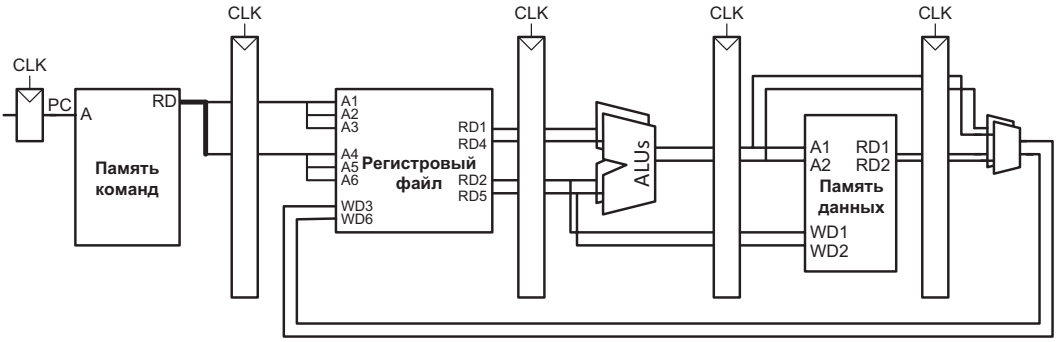


Рис. 7.68 Тракт данных суперскалярного процессора

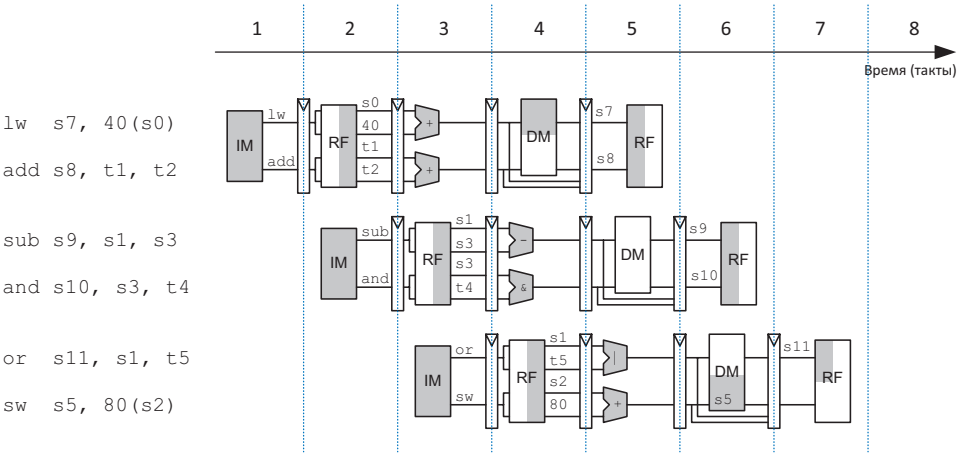


Рис. 7.69 Работающий суперскалярный конвейер

Процессор RISC-V SweRV EH1, разработанный и выпускаемый Western Digital, представляет собой двухконвейерное суперскалярное ядро.

Выполнять много команд одновременно довольно трудно из-за зависимостей между ними. Рассмотрим **рис. 7.70**, на котором показана диаграмма конвейера, выполняющего программу с зависимостями между данными. Зависимости в коде показаны голубым цветом. Команда `add` зависит от `s8`, значение которого изменяет команда `lw`, поэтому эти команды нельзя запускать на выполнение одновременно. На самом деле команда `add` приостанавливается еще на один такт, чтобы `lw` могла переслать через байпас прочитанное из памяти значение `s8` команде `add` на пятом такте. Оставшиеся конфликты (между `sub` и `and` из-за `s8` и между `or` и `sw` из-за `s11`) разрешаются при помощи пересылки результатов через байпас. Эта программа, показанная ниже, выполняется за пять тактов, а IPC = 6/5 = 1,2.

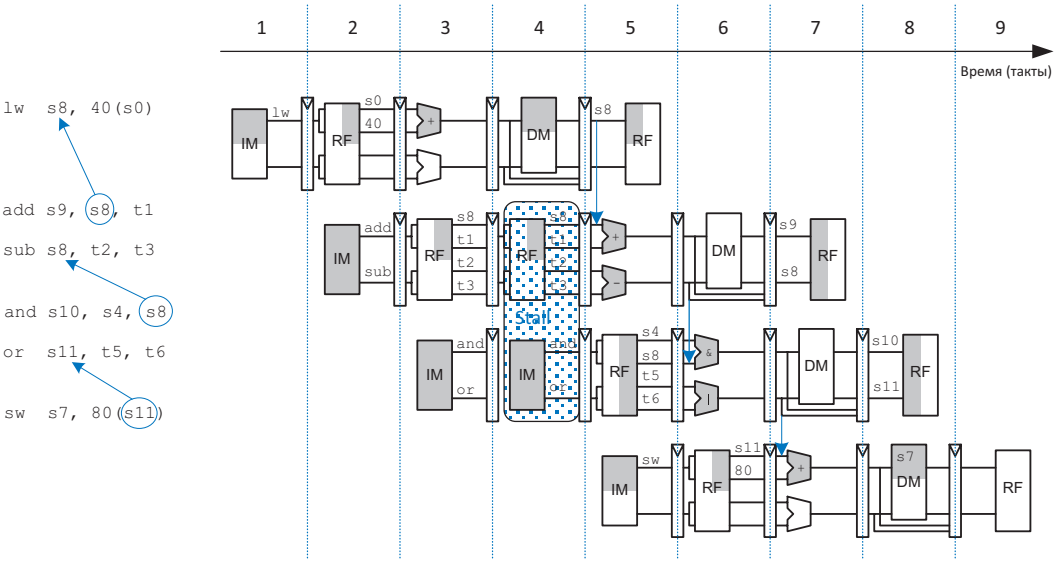


Рис. 7.70 Программа с зависимостями между командами

Наш конвейерный процессор RISC-V — это скалярный процессор. Векторные процессоры широко применялись в суперкомпьютерах в 1980-х и 1990-х годах, потому что они эффективно обрабатывали длинные векторы данных, которые часто встречаются в научных вычислениях, а теперь они активно используются в графических процессорах. Векторная обработка (и SIMD в целом) является примером параллелизма на уровне данных, когда несколько данных могут обрабатываться параллельно. Современные высокопроизводительные процессоры являются суперскалярными, поскольку выполнение нескольких параллельных команд гибче, чем обработка векторов. При этом в состав современных процессоров обычно входит модуль SIMD для обработки коротких векторов данных, которые встречаются в мультимедийных и графических приложениях. Архитектура RISC-V содержит векторное расширение (V) для поддержки векторных операций.

Вспомните, что у параллелизма две формы — временная и пространственная. Конвейерная реализация — это пример временного параллелизма, а наличие нескольких экземпляров одних и тех же функциональных блоков — пример пространственного. Суперскалярные процессоры используют обе формы параллелизма, чтобы достичь производительности, значительно превосходящей производительность наших однотактного и многотактного процессоров.

Коммерческие суперскалярные процессоры могут иметь три-, четыре- или даже шесть параллельных конвейеров. Им приходится обрабатывать и конфликты управления, вызываемые, например, условными переходами, и конфликты данных. К сожалению, в реальных программах встречается много зависимостей, поэтому суперскалярные процессоры с большим количеством каналов редко могут использовать все свои функциональные блоки полностью. Более того, большое количество функциональных блоков и сложности с организацией байпаса требуют множества дополнительных логических элементов и потребляют большое количество электроэнергии.

7.7.5. Процессор с внеочередным выполнением команд

Чтобы справиться с проблемой зависимостей, *процессор с внеочередным выполнением команд* (out-of-order processor) заранее просматривает большое количество команд, которые, по его мнению, надо будет скоро выполнить, чтобы как можно быстрее обнаружить и запустить на выполнение те команды, которые не зависят друг от друга. Команды могут выполняться не в том в порядке, в котором они находятся в программе, но только при условии, что процессор учитывает все зависимости, что позволит программе выдать ожидаемый результат.

Рассмотрим выполнение той же программы, которую мы привели на **рис. 7.70**, на двухконвейерном суперскалярном процессоре с внеочередным выполнением команд. За один такт процессор может запускать до двух команд из любой части программы при условии, что он соблюдает все зависимости. На **рис. 7.71** показаны зависимости данных и работа процессора.

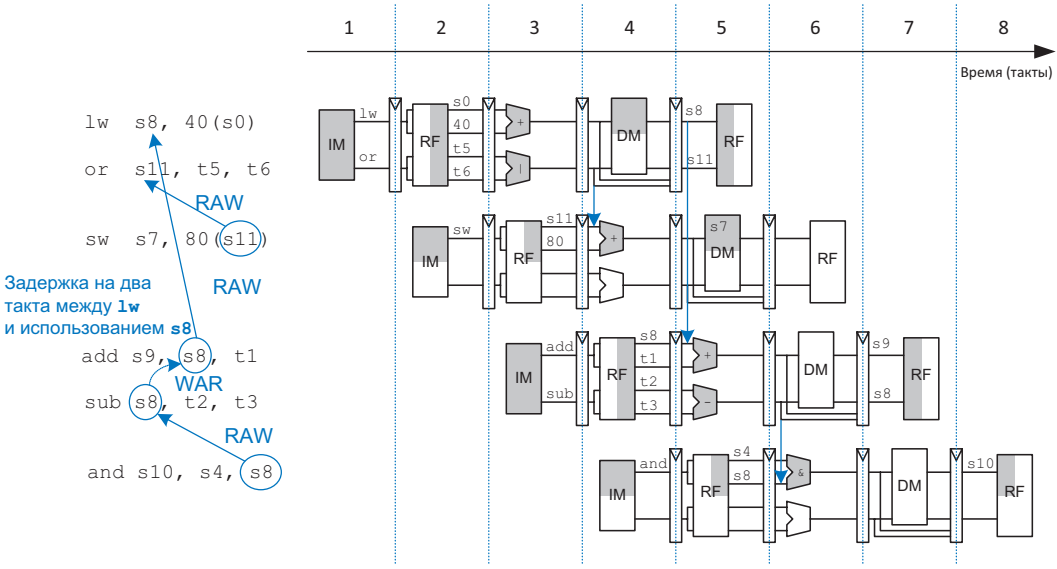


Рис. 7.71 Внеочередное выполнение команд, зависящих друг от друга

Чуть позже мы обсудим классификацию зависимостей (RAW и WAR). Ниже описаны ограничения на запуск команд.

Такт 1

- Команда `lw` запускается на выполнение.
- Команды `add`, `sub` и `and` зависят от `lw`, так как используют `s8`, поэтому их пока запустить нельзя. А вот команда `or` не зависит от `lw`, поэтому она тоже запускается на выполнение.

Такт 2

- Вспомните, что между запуском команды `lw` и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому `add` запустить пока нельзя, так как она зависит от `s8`. Команда `sub` записывает результат в `s8`, поэтому ее нельзя запускать перед `add`, иначе `add` получит неверное значение `s8`. Команда `and` зависит от `sub`.
- Запускается только команда `sw`.

Такт 3

- На третьем такте значение в `s8` становится корректным, поэтому запускается `add`. Команда `sub` тоже запускается на выполнение, потому что `add` прочитает `s8` раньше, чем `sub` изменит его.

Такт 4

- Запускается команда `and`. Значение `s8` пересылается от `sub` к `and` через байпас.

Таким образом, процессор с внеочередным выполнением команд запускает шесть команд за четыре такта, то есть его IPC = 6/4 = 1,5. Зависимость `add` от `lw` из-за использования `s8` называется *конфликтом чтения после записи*, или RAW-конфликтом (read-after-write, RAW). Команда `add` не имеет права читать регистр `s8` до тех пор, пока `lw` в него не запишет данные. Мы уже встречались с таким типом зависимости, когда рассматривали конвейерный процессор, и знаем, как с ней справляться. Такая зависимость по своей природе ограничивает скорость выполнения программы, даже если у процессора есть бесконечно много функциональных блоков для выполнения команд. Аналогично зависимость `sw` от `or` из-за использования регистра `s11` и зависимость `and` от `sub` из-за использования регистра `s8` являются RAW-зависимостями.

Зависимость между `sub` и `add` из-за использования `s8` называется *конфликтом записи после чтения*, или WAR-конфликтом (write-after-read, WAR), или *антизависимостью* (antidependence). Команда `sub` не имеет права записывать в регистр `s8` до того, как `add` прочитает его. Это необходимо, чтобы команда `add` получила правильное значение в соответствии с исходным порядком команд в программе. WAR-конфликты не могут возникнуть в обычной реализации конвейера, но могут возникнуть

в процессоре с внеочередным выполнением команд, если он попытается запустить зависимую команду (в данном случае sub) слишком рано.

В отличие от RAW-конфликтов, WAR-конфликты не являются неизбежными во время работы программы. Это просто следствие решения программиста (или компилятора) использовать один и тот же регистр для двух не связанных друг с другом команд. Если бы команда sub записывала результат в s3 вместо s8, то зависимость исчезла бы, и можно было бы запустить sub перед add. Архитектура RISC-V имеет только 31 регистр, поэтому иногда программист вынужден повторно использовать регистр и создавать опасность конфликтов только потому, что все остальные регистры уже используются.

Третий тип конфликтов, не показанный в программе, называется *конфликтом записи после записи*, или WAW-конфликтом (write-after-write, WAW). Его еще называют *зависимостью вывода* (output dependency) или *ложной зависимостью* (false dependency). WAW-конфликт случается, когда команда пытается записать значение в регистр после того, как это сделала следующая по ходу программы команда. В результате этого конфликта в регистр будет записано неверное значение. Например, в программе ниже две команды, lw и add, записывают результат своего выполнения в s7. Согласно порядку команд в программе, окончательное значение в s7 должна записать команда add. Если бы процессор с внеочередным выполнением команд попытался выполнить add перед lw, то произошел бы WAW-конфликт.

```
lw s7, 0(t3)
add s7, s1, t2
```

Как и WAR-конфликты, WAW-конфликты также не являются неизбежной ситуацией. Они, опять же, возникают из-за решения программиста (или компилятора) использовать один и тот же регистр для двух не связанных друг с другом команд. Если команда add была запущена первой, процессор мог бы устранить WAW-конфликт, запретив команде lw записывать свой результат в s7. Этот прием называют «предотвращением» (squashing) выполнения команды lw[1].

Процессоры с внеочередным выполнением команд используют специальную таблицу, чтобы отслеживать команды, ожидающие запуска. Эта таблица, иногда называемая таблицей готовности (scoreboard), содержит информацию о зависимостях тех команд, которые процессор толь-

[1] Вы можете спросить, зачем вообще нужно запускать команду add? Дело в том, что процессоры с внеочередным выполнением команд обязаны гарантировать, что во время выполнения программы произойдут все те же исключения, которые произошли бы, если бы все команды выполнялись в исходном порядке. Команда add может потенциально вызвать исключение из-за арифметического переполнения, поэтому ее необходимо запустить, для того чтобы проверить, произойдет переполнение или нет, даже если ее результат будет не нужен.

ко собирается запустить. Размер таблицы определяет, сколько команд одновременно могут являться кандидатами на запуск. На каждом такте процессор сверяется с таблицей и запускает столько команд, сколько он может, с учетом зависимостей и количества доступных функциональных блоков (АЛУ, портов памяти и т. д.).

Параллелизм на уровне команд (instruction level parallelism, ILP) – это количество команд конкретной программы, которые могут одновременно выполняться на определенной микроархитектуре. Теоретические исследования показали, что ILP для микроархитектур с внеочередным выполнением команд, при условии идеального предсказания переходов и огромного количества функциональных блоков, может быть довольно высоким. К сожалению, на практике даже шестиканальные суперскалярные процессоры с внеочередным выполнением команд редко достигают ILP, превышающего 2 или 3.

7.7.6. Переименование регистров

Для того чтобы устранить WAR-конфликты, процессоры с внеочередным выполнением команд используют прием, который называется *переименованием регистров* (register renaming). Для реализации переименования регистров в процессор добавляют так называемые неархитектурные (невидимые программисту) *регистры переименования* (renaming registers). Например, в процессор RISC-V может быть добавлено 20 регистров переименования, называемых `r0-r19`. Программист не может использовать эти регистры напрямую, потому что они не являются частью архитектуры процессора. Но при этом процессор может использовать их для устранения конфликтов.

Например, в предыдущем разделе был показан WAR-конфликт между командами `sub s8, t2, t3` и `add s9, s8, t1`, который случился из-за повторного использования `s8`. Процессор с внеочередным выполнением команд может переименовать `s8` в `r0` для команды `sub`. После этого `sub` можно выполнить быстрее, потому что у `r0` нет зависимости от команды `add`. У процессора есть таблица с информацией о том, какие регистры были переименованы, поэтому он может соответствующим образом переименовать регистры и в последующих зависимых командах. В этом примере `s8` также необходимо переименовать в `r0` в команде `and`, поскольку она использует результат команды `sub`. На **рис. 7.72** показана та же программа, что и на **рис. 7.71**, но выполняемая на процессоре с переименованием регистров. Чтобы устранить WAR-конфликт, регистр `s8` был переименован в `r0` в командах `sub` и `add`. Ниже описаны ограничения на запуск команд.

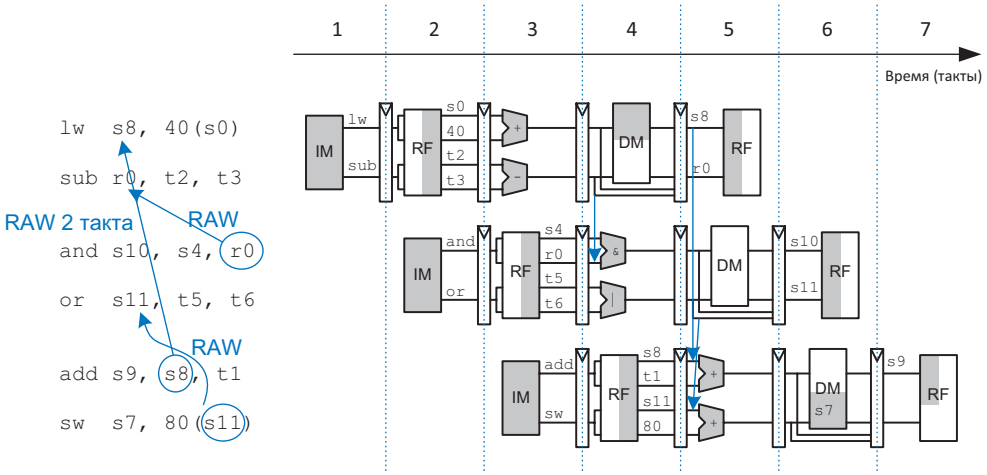


Рис. 7.72 Внеочередное выполнение команд с переименованием регистров

Такт 1

- Команда `lw` запускается на выполнение.
- Команда `add` зависит от `lw` из-за использования `s8`, поэтому пока что запустить ее нельзя. При этом команда `sub` теперь независима, так как ее регистр результата переименован в `r0`, поэтому `sub` также запускается на выполнение.

Такт 2

- Вспомните, что между запуском команды `lw` и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому `add` запустить пока нельзя, так как она зависит от `s8`. Команда `and` зависит от уже запущенной на выполнение `sub`, поэтому ее тоже можно запускать – значение `r0` будет передано от `sub` к `and` через байпас.
- У команды `or` нет зависимостей, поэтому она тоже запускается.

Такт 3

- На третьем такте значение в `s8` становится корректным, поэтому запускается `add`.
- Значение в `s11` также становится корректным, поэтому запускается и команда `sw`.

Таким образом, процессор с внеочередным выполнением команд и переименованием регистров запускает шесть команд за три такта, то есть его IPC равно 2.

7.7.7. Многопоточность

Так как *параллелизм на уровне команд* (instruction level parallelism, ILP) у реальных программ, как правило, довольно низок, добавление все новых и новых функциональных блоков к суперскалярному процессору или процессору с внеочередным выполнением команд приводит ко все меньшему эффекту. Еще одной проблемой является то, что основная память гораздо медленнее, чем процессор (мы рассмотрим это в **главе 8**). Большинство команд чтения из памяти и записи в память данных работают со значительно более быстрой и маленькой памятью, которая называется *кеш-памятью* (cache memory). К сожалению, если нужных команд или данных в кеше нет, то процессор может быть приостановлен на 100 и более тактов, ожидая получения информации из основной памяти. Многопоточность – это способ загрузить работой процессор с большим количеством функциональных блоков, даже если у программы низкий ILP или она приостановлена на время ожидания данных из памяти.

Для того чтобы объяснить суть многопоточности, нам надо определить несколько новых терминов. Программа, которая выполняется на компьютере, называется *процессом* (process). Компьютеры могут выполнять несколько процессов одновременно. Например, на своем ПК вы можете слушать музыку и работать в сети интернет, одновременно запустив антивирус. Каждый процесс состоит из одного или более *потоков команд* (threads), которые тоже выполняются одновременно. Например, в текстовом редакторе один поток может обрабатывать набор текста пользователем, второй поток в это время может проверять орфографию, третий – печатать документ на принтере. При такой организации пользователю не нужно ждать, пока закончится печать на принтере, чтобы снова набирать текст. Степень, до которой процесс можно разделить на несколько одновременно выполняющихся потоков, определяет *уровень параллелизма на уровне потоков* (thread level parallelism, TLP).

В обычном процессоре одновременная работа потоков – это только иллюзия. Реально же потоки выполняются процессором по очереди под управлением операционной системы. Когда «смена» одного потока подходит к концу, ОС сохраняет его архитектурное состояние, загружает из памяти архитектурное состояние следующего потока и передает ему управление. Эта процедура называется *переключением контекста* (context switching). До тех пор, пока процессор переключается между потоками достаточно быстро, пользователю кажется, что все потоки выполняются одновременно. В архитектуре RISC-V один из 32 регистров, регистр указателя потока `tp` (thread pointer, `x4`), выделен для указания (хранения адреса) локальной памяти потока.

У *аппаратного многопоточного процессора* есть несколько копий архитектурного состояния, вследствие чего несколько потоков могут быть активны одновременно. Например, если мы расширим процессор так, чтобы у него было четыре счетчика команд и 128 регистров, то одновременно могут быть доступны четыре потока. Если один из них приостанавливается в ожидании данных из основной памяти, то процессор немедленно переключает контекст на другой поток. Это переключение происходит безо всяких накладных расходов, так как счетчик команд и регистры уже доступны и их не надо отдельно загружать. Более того, если один из потоков не может в полной мере использовать все функциональные блоки процессора из-за недостаточного уровня параллелизма, то другой поток может запустить на исполнение команды, использующие незанятые блоки. Переключение между потоками может быть мелкозернистым или крупнозернистым. *Мелкозернистая многопоточность* представляет собой переключение между потоками в каждой команде и нуждается в аппаратной поддержке многопоточности. *Крупнозернистая многопоточность* переключает поток только при дорогостоящих остановках, таких как длительные обращения к памяти из-за отсутствия нужных данных в кеше.

Многопоточность не улучшает производительность отдельного потока, потому что она не повышает ILP. Тем не менее она улучшает общую пропускную способность процессора, так как несколько потоков могут более полно использовать те ресурсы процессора, которые бы не использовались при выполнении одного-единственного потока. Многопоточность относительно легко реализовать, так как требуется добавить только копии счетчика команд и регистрового файла. Функциональные блоки и память копировать не надо.

7.7.8. Мультипроцессоры

(*При участии Мэтью Уоткинса*)

Современным процессорам доступно неимоверное количество транзисторов. Если использовать эти транзисторы только для того, чтобы увеличивать длину конвейера или добавлять функциональные блоки в суперскалярный процессор, то в определенный момент рост производительности станет незначительным, а энергопотребление станет слишком большим. Примерно в 2005 году разработчики компьютерных микроархитектур начали размещать по нескольку копий процессора на одном чипе; эти копии называются ядрами.

Многопроцессорная система (multiprocessor system), или просто *мультипроцессор*, состоит из нескольких процессоров и аппаратуры для соединения их между собой. Мультипроцессоры можно разделить на три основных класса: *симметричные* (или гомогенные) мультипроцессоры, *асимметричные* (гетерогенные) мультипроцессоры и *кластеры*.

Симметричные мультипроцессоры

Симметричные мультипроцессоры состоят из двух или более идентичных процессоров, совместно использующих одну основную память. Эти процессоры могут быть отдельными микросхемами или несколькими ядрами в одном корпусе микросхемы.

Мультипроцессоры можно использовать или для того, чтобы выполнять больше потоков одновременно, или для того, чтобы быстрее выполнять один конкретный поток. Выполнять больше потоков одновременно довольно просто – эти потоки можно просто распределить между процессорами. К сожалению, пользователям персональных компьютеров обычно не нужно выполнять много потоков – им нужно выполнять всего несколько потоков, но при этом максимально быстро. Ускорение одного потока при помощи мультипроцессора – далеко не тривиальная задача. Чтобы достичь этого, программист должен разделить один медленный поток на несколько меньших потоков, которые можно будет запустить на разных процессорах. Все еще больше усложняется, если процессорам нужно обмениваться данными. В настоящее время эффективное использование большого количества процессорных ядер – одна из главных проблем, стоящих перед разработчиками компьютеров и программистами.

Симметричные мультипроцессоры имеют ряд преимуществ. Их относительно просто разработать, поскольку процессор можно разработать один раз, а затем многократно дублировать для повышения производительности. Программирование и выполнение кода на симметричном мультипроцессоре также относительно просты, потому что любая программа может выполняться на любом процессоре в системе и при этом обеспечивать примерно одинаковую производительность.

Гетерогенные мультипроцессоры

К сожалению, нет никакой гарантии, что с добавлением все большего и большего количества ядер производительность системы продолжит увеличиваться. Большинство пользовательских программ используют лишь несколько потоков в каждый момент времени, а у среднестатического пользователя обычно запущено не более 2–3 программ, работающих одновременно. И хотя этого достаточно, чтобы загрузить двух- или четырехъядерную систему, но добавление большего числа ядер будет приводить ко все менее заметным результатам до тех пор, пока программы не начнут использовать параллелизм более широко. Кроме этого, так как процессоры общего назначения разрабатываются с целью обеспечивать хорошую среднюю производительность на широком классе задач, то они, как правило, являются далеко не самым энергоэффективным способом выполнения той или иной конкретной операции. Энергоэффек-

тивность особенно важна в мобильных системах, где энергопотребление сильно ограничено.

Гетерогенные мультипроцессоры (heterogeneous multiprocessors), также называемые асимметричными, призваны решить эти проблемы путем использования различных типов процессорных ядер и/или специализированной аппаратуры в одной системе. Каждое приложение использует те ресурсы, которые позволяют достичь или наилучшей производительности, или наилучшего соотношения производительности и энергопотребления для этого конкретного приложения. Так как в наши дни разработчики могут использовать сколько угодно транзисторов, то никого особенно не заботит, что не каждое приложение сможет использовать все имеющиеся в наличии аппаратные блоки.

Гетерогенные системы могут принимать разные формы. Они могут включать в себя ядра с различными микроархитектурами, имеющими разное соотношение энергопотребления, производительности и занимаемой на чипе площади. Архитектура RISC-V была специально разработана для ряда реализаций, от недорогих встроенных процессоров до высокопроизводительных мультипроцессоров. Другая стратегия построения гетерогенных систем – это использование *ускорителей*, когда система содержит специальное оборудование, оптимизированное по производительности или энергоэффективности при выполнении определенных типов задач. Например, современная мобильная система на кристалле (SoC) может содержать отдельные ускорители для обработки графики, видео, беспроводной связи, задач реального времени и криптографии. Эти ускорители могут быть от 10 до 100 раз эффективнее (по производительности, стоимости и площади), чем универсальные процессоры, решающие аналогичные задачи. Еще один особый класс ускорителей представляют *цифровые сигнальные процессоры*. Эти процессоры имеют специальный набор команд, оптимизированный для математических задач цифровой обработки сигналов.

У гетерогенных систем есть и недостатки. Они сложнее и в разработке, и в программировании, так как требуется не только разработать разнообразные аппаратные блоки, но и решить, когда и как наилучшим образом использовать различные ресурсы системы. Таким образом, у гомогенных и у гетерогенных систем есть свои ниши применения. Гомогенные мультипроцессоры подходят, например, для больших центров обработки данных, где нет недостатка в задачах с высоким параллелизмом на уровне потоков. Гетерогенные системы хороши в случае более разнообразной вычислительной нагрузки и ограниченного параллелизма.

Кластеры

В отличие от других мультипроцессоров, у каждого процессора в кластерных многопроцессорных системах есть своя отдельная подсистема

локальной памяти, и он не делит ее с другими процессорами. Кластером также называют группу обычных персональных компьютеров, соединенных вместе сетью и выполняющих специализированные программы, позволяющие компьютерам вместе решать масштабную задачу. Входящие в кластер компьютеры обмениваются данными по сети, а не через общую память. Наиболее известные и мощные кластеры – это *центры обработки данных* (ЦОД), в которых смонтированные в огромных ангарах стойки компьютеров и накопителей данных объединены в сеть и совместно используют системы питания и охлаждения. Такие системы обычно состоят из 50 000–100 000 компьютеров или серверов и стоят от 150 млн долларов[1]. Инициаторами строительства ЦОД выступили такие интернет-гиганты, как Google, Amazon и Facebook, которые нуждаются в поддержке миллионов пользователей по всему миру. Одним из основных преимуществ подобных кластеров является возможность безболезненно заменять отдельные компьютеры по мере необходимости из-за их поломок или для обновления.

В последние годы традиционные серверы, принадлежащие различным компаниям, заменяются облачными вычислениями, когда небольшая компания арендует часть ресурсов ЦОД у таких компаний, как Google и Amazon. Аналогичным образом вместо приложения, полностью работающего на персональном мобильном устройстве, таком как смартфон или планшет, часть приложения может работать в облаке, на мощных серверах, тем самым ускоряя вычисления и делая хранение данных более эффективным. Такой подход называется «*программное обеспечение как услуга*» (software as a service, SaaS). Типичным примером SaaS является интернет-поиск, когда база данных хранится в ЦОД провайдера поисковой системы. Компаниям, арендующим облачные или веб-сервисы, требуется как конфиденциальность (изоляция от другого программного обеспечения, работающего в этом же облаке), так и производительность. Для этого клиенту предоставляют *виртуальную машину*, которая имитирует обычный компьютер, включая его операционную систему, хотя физическая машина, на которой запущена виртуальная машина, может работать под управлением другой операционной системы. На одной физической машине могут одновременно работать несколько виртуальных машин, при этом использование общих физических ресурсов, таких как память и система ввода-вывода, может быть разделено во времени. Это позволяет поставщикам услуг облачных вычислений, таким как Amazon Web Services (AWS), эффективно использовать физические ресурсы, обеспечивать взаимную изоляцию виртуальных машин и переносить виртуальные машины с неисправных или низкоэффективных компьютеров. *Гипервизор*, также называемый *монитором виртуальных машин* (virtual machine monitor, VMM), –

[1] D. Patterson, J. Hennessy, Computer Organization and Design, The Hardware-Software Interface: RISC-V Edition, Morgan Kaufmann, © 2018.

это программное обеспечение, которое запускает виртуальную машину и выделяет ей физические ресурсы. Гипервизор выполняет функции, обычно присущие операционной системе, такие как управление системой ввода-вывода, ресурсами центрального процессора и памятью. В структуре системы гипервизор располагается между *хостом* (физической аппаратной платформой) и операционной системой, которую он эмулирует. Архитектуры набора команд, которые позволяют гипервизору работать непосредственно на оборудовании (в отличие от программного обеспечения), называются *виртуализируемыми*. Подобные архитектуры позволяют создавать более эффективные и высокопроизводительные виртуальные машины. Примерами виртуализируемых архитектур являются x86, RISC-V и IBM 370. Архитектуры ARMv7 и MIPS не виртуализируются, но ARM представила в 2013 году расширение для виртуализации вместе с новой архитектурой ARMv8.

Облачные вычисления также являются важной частью приложений интернета вещей (IoT), в которых устройства, такие как умные колонки, телефоны и датчики, подключаются через сеть, например Bluetooth или Wi-Fi. В качестве примера можно назвать подключение наушников к смартфону с помощью Bluetooth или подключение голосовых помощников Alexa или Siri с помощью соединения Wi-Fi. Недорогие устройства (наушники, Google Home для Google Assistant или Echo Dot для Alexa) подключаются через сеть к мощным серверам, которые могут транслировать музыку или, в случае Siri и Alexa, выполнять распознавание речи, запрашивать поисковые базы данных и непосредственно выполнять вычисления.

7.8. Живой пример: эволюция микроархитектуры RISC-V

В этом разделе мы проследим за развитием микроархитектуры RISC-V с момента создания RISC-V в 2010 году. Поскольку полное описание базового набора команд завершено совсем недавно, лишь немногие из чипов RISC-V, разработку которых начали в 2017 году, дойдут до коммерческой реализации в 2021–2022 гг. Но мы ожидаем, что эта ситуация быстро исправится по мере появления новых вспомогательных инструментов и сред разработки.

Большинство существующих реализаций RISC-V представляют собой маломощные или встраиваемые процессоры, но ожидается скорое появление и более мощных реализаций данной архитектуры. Сообщество RISC-V

Хотя RISC-V — это архитектура с открытым исходным кодом, а не микроархитектура, регулярно появляются новые реализации аппаратуры с открытым исходным кодом, включая ядра SweRV Western Digital, SoC SweRVolf и платформу PULP (Parallel Ultra Low Power, параллельное сверхнизкое энергопотребление). Постоянно растущее количество аппаратных реализаций и вспомогательных инструментов RISC-V называется *экосистемой* RISC-V. В предисловии рассказано о том, как получить и использовать некоторые инструменты и реализации процессора на HDL с открытым исходным кодом для разработки RISC-V.

Исходный код ядер Western Digital (формат SystemVerilog) с открытым исходным кодом доступен по адресу: https://github.com/chipsalliance/Cores-SweRV.

Роберт Голла — старший научный сотрудник Western Digital, отвечавший за разработку архитектуры процессорных ядер EH1, EL2 и EH2 RISC-V с открытым исходным кодом для встроенных процессоров Western Digital. Он также разработал ядра Oracle T4, M7, M8 и M9 с внеочередным выполнением команд, многопоточное ядро Sun N2 и встраиваемое ядро e500 от Motorola и внес свой вклад в создание нового поколения процессоров Cyrix x86 M3, маломощных 603 и 603e от NXP, а также POWER1 и IBM POWER1, POWER2 от IBM. Автор более 50 патентов, связанных с разработкой микропроцессоров.

International (riscv.org) публикует постоянно растущий список ядер и платформ SoC. Коммерческие ядра RISC-V используются, среди прочего, в платах разработки SiFive HiFive, жестких дисках Western Digital и графических процессорах NVIDIA.

По состоянию на 2021 год двумя наиболее популярными коммерческими процессорами RISC-V являются ядро SiFive Freedom E310 и ядро SweRV с открытым исходным кодом, разработанное Western Digital, которое представлено в трех версиях. Freedom E310 – это недорогой встраиваемый процессор, применяемый в платах разработки SiFive HiFive и Sparkfun RED-V. Он поддерживает набор команд RV32IMAC (RV32I с умножением/делением [M], атомарным доступом к памяти [A] и поддержкой компактных инструкций [C]) и имеет 8 КБ памяти программ, 8 КБ маскируемой ROM для кода загрузчика, 16 КБ SRAM памяти данных и двувходовый ассоциативный кеш команд размером 16 КБ. Он также поддерживает интерфейсы JTAG, SPI, I2C, UART и флеш-память QSPI. Процессор работает на частоте 320 МГц и представляет собой ядро с последовательным выполнением операций и 5-ступенчатым конвейером, который работает в соответствии со стадиями, рассмотренными в этой главе. На **рис. 7.73** показана блок-схема процессора FE310-G002, установленного на плате HiFive 1 Rev B.

Ядро Western Digital SweRV поставляется в трех версиях с открытым исходным кодом: EH1, EH2 и EL2. Версия EH1 – это 32-битное двустороннее суперскалярное ядро с девятиступенчатым конвейером и ограниченной поддержкой внеочередного выполнения. Все три ядра реализуют набор инструкций RV32IMC, который содержит 32-битный базовый набор инструкций, а также расширения для работы с компактными инструкциями (C) и операциями умножения/деления (M). При использовании 28-нм техпроцесса его тактовая частота составляет 1 ГГц. HDL-код ядра также можно реализовать на FPGA. В ядро EH2 добавлена двухпотоковая обработка. Ядро EL2 – это процессор с относительно небольшой производительностью, предназначенный для встраиваемых систем. На **рис. 7.74** показаны девять стадий конвейера ядра EH1, которые начинаются с двух стадий выборки, одной стадии выравнивания (align) и одной стадии декодирования. На стадии декодирования обрабатываются до двух инструкций. После этого конвейер разделяется на пять параллельных трактов: тракт чтения из памяти/записи в память, два тракта для целочисленных инструкций (таких как

add, sub и xor), один тракт для умножения и один для деления – то есть инструкции выполняются не в порядке поступления в конвейер из-за их задержки в 34 такта. Последние две стадии конвейера – это подтверждение (commit) и запись результата. Стадия подтверждения требуется из-за нарушения порядка выполнения операций и записи регистров в буфер. На последней стадии при необходимости происходит запись результатов в регистровый файл.

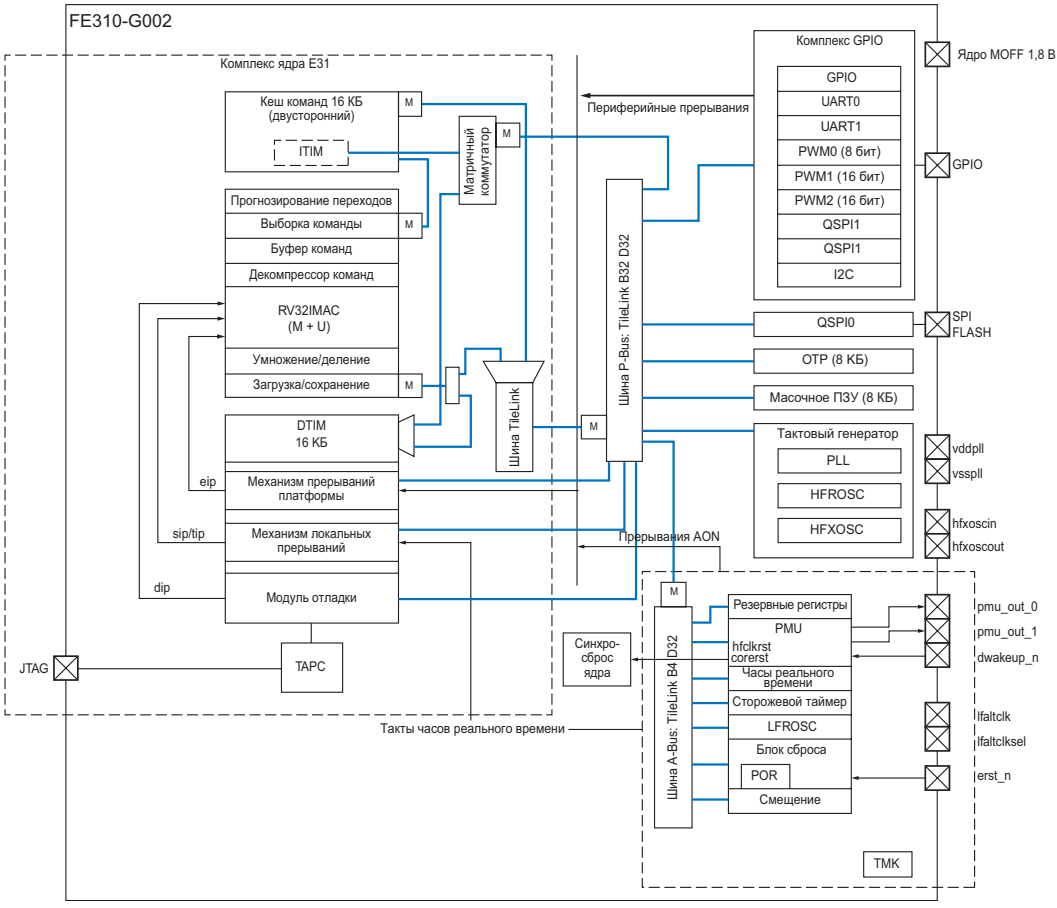


Рис. 7.73 Блок-схема процессора Freedom E310-G002
(предоставлена SiFive Inc., SiFive FE310-G002 Preliminary Datasheet v1p0, 2019)

На **рис. 7.75** изображена блок-схема SweRV EH1. Она содержит собственно процессор (на рисунке обозначен как ядро SweRV EH1), кеш инструкций (I-Cache), память данных и инструкций (DCCM и ICCM – память данных и команд), программируемый контроллер прерываний (PIC), интерфейс JTAG, интерфейсы доступа к памяти и отладки, кото-

Пройдя бесплатный курс RISC-V FPGA (RVfpga) от компании Imagination Technologies, вы узнаете, как реализовать ядро SweRV EH1 на FPGA, и научитесь компилировать и запускать программы на ядре RISC-V. В состав курса входят лабораторные работы и упражнения, которые на практике демонстрируют добавление периферийных устройств в ядро, работу конвейера, предсказание переходов ветвлений и иерархию памяти, а также использование таймеров, прерываний и счетчиков производительности. Эти лабораторные работы и материалы находятся в свободном доступе на странице университетской программы Imagination Technologies University по адресу https://university.imgtec.com/rvfpga/.

рые могут быть сконфигурированы как шины AXI4 или AHB-Lite. Процессор состоит из блоков выборки команд (IFU), декодирования (DEC), выполнения (EXU) и чтения из памяти / записи в память (LSU). Блок IFU охватывает обе стадии конвейерной выборки; DEC выполняет этапы выравнивания и декодирования; EXU выполняет работу на всех остальных стадиях, кроме конвейера чтения из памяти / записи в память, который расположен в блоке LSU. В систему входит четырехвходовый ассоциативный кеш команд, объем которого можно настроить в диапазоне от 16 до 256 КБ. Блоки DCCM и ICCM называются *тесно связанной памятью* (closely coupled), поскольку они представляют собой встроенную память с малой задержкой, которую можно сконфигурировать на объем от 4 до 512 КБ.

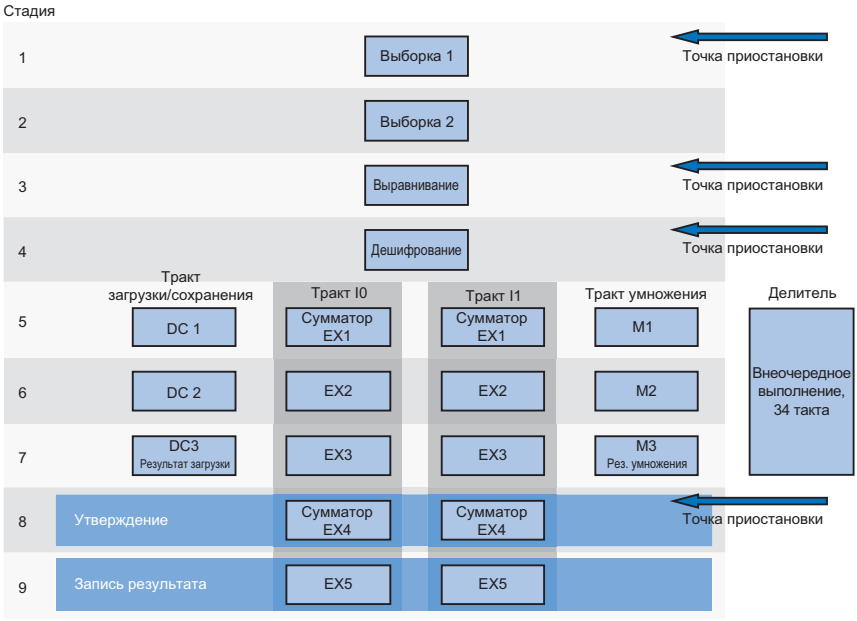


Рис. 7.74 9-стадийный конвейер ядра SweRV EH1
(рисунок предоставлен компанией Western Digital Corporation, Справочное руководство программиста RISC-V SweRVTM EH1, 2020)

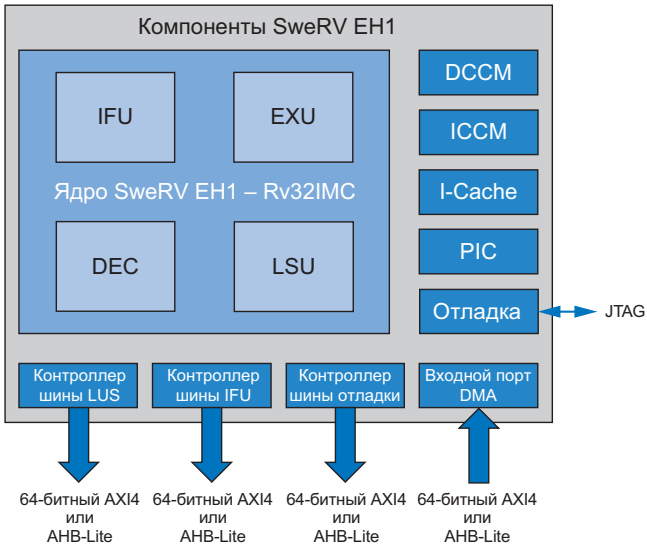


Рис. 7.75 Компоненты SweRV EH1
(рисунок предоставлен компанией Western Digital Corporation, Справочное руководство программиста RISC-V SweRVTM EH1, 2020)

7.9. Заключение

В этой главе мы рассмотрели три способа построения процессоров, отличающихся разным соотношением производительности, занимаемых ресурсов чипа и стоимости. Мы считаем, что это сродни магии – как может столь сложное на вид устройство, как микропроцессор, на самом деле быть столь простым, что его схема занимает всего полстраницы? Более того, принцип его работы, такой таинственный для непосвященных, на поверку оказывается довольно очевидным.

Чтобы разобраться с микроархитектурой, нам понадобились знания почти изо всех разделов, ранее представленных в этой книге. Мы разрабатывали комбинационные и последовательностные схемы так, как было описано в **главе 2** и **главе 3**, применяли строительные блоки, рассмотренные в **главе 5**. Мы воплощали в жизнь архитектуру RISC-V, описанную в **главе 6**. Используя методы, представленные в **главе 4**, мы смогли описать микроархитектуру всего с помощью нескольких страниц HDL-кода.

Разработка различных вариантов микроархитектур также потребовала от нас применения принципов управления сложностью. Микроархитектурная абстракция образует связь между логическим и архитектурным уровнями абстракции и представляет собой главную тему данной

книги о разработке цифровых систем и компьютерной архитектуре. Мы также использовали абстракции уровня блочных диаграмм и языков описания аппаратуры, чтобы в сжатой форме описывать взаимное расположение компонентов. При разработке микроархитектур мы широко применяли принципы повторяемости и модульности, повторно используя библиотеки часто используемых строительных блоков, таких как АЛУ, блоки памяти, мультиплексоры и регистры. Мы активно использовали иерархический подход, разделив микроархитектуру на тракт данных и устройство управления, которые мы создали из функциональных блоков, а те, в свою очередь, – из логических элементов, а логические элементы – из транзисторов, как было описано в первых пяти главах.

В этой главе мы сравнили однотактную, многотактную и конвейерную микроархитектуры процессора RISC-V. Все они реализуют одно и то же подмножество набора команд RISC-V и имеют одинаковое архитектурное состояние. Однотактный процессор наиболее прост, а каждая его команда выполняется за один такт, т. е. его CPI равен 1.

Многотактный процессор выполняет команды за переменное количество более коротких этапов. Таким образом он может использовать одно-единственное АЛУ вместо нескольких сумматоров. Но ему требуется несколько неархитектурных регистров, чтобы хранить промежуточные результаты вычислений между этапами. В теории многотактный процессор мог бы быть быстрее, так как не все команды выполняются за одинаковое время. Но на практике он обычно медленнее, так как длительность его такта ограничена длительностью самого медленного этапа тракта данных, на который, в свою очередь, негативно влияют накладные расходы, связанные с использованием неархитектурных регистров.

Конвейерный процессор разделяет однотактный процессор на пять относительно быстрых стадий. Для этого между его стадиями добавлены временные регистры, что позволило изолировать друг от друга пять одновременно выполняющихся команд. В идеальном мире его CPI был бы равен 1, но конфликты в конвейере приводят к необходимости периодически приостанавливать и очищать его, что увеличивает CPI. Логика, необходимая для разрешения конфликтов, также увеличивает сложность процессора. Период тактового сигнала мог бы быть в пять раз меньше, чем у однотактного процессора, но на практике он далеко не так мал, потому что ограничен скоростью работы самой медленной стадии, а также накладными расходами из-за добавленных между стадиями регистров. Тем не менее конвейерная обработка обеспечивает существенное увеличение производительности, поэтому она используется во всех современных высокопроизводительных микропроцессорах.

Хотя микроархитектуры, рассмотренные в этой главе, реализуют только ограниченное подмножество команд архитектуры RISC-V, мы показали, что добавление новых команд требует внесения весьма простых

и понятных изменений в тракт данных и устройство управления. Поддержка исключений также требует лишь незначительных изменений.

Существенным ограничением этой главы является то, что мы считали подсистему памяти идеальной, обеспечивающей быстрый доступ и способность хранить всю программу и все данные целиком. В реальности же большая и быстрая память очень дорогая. В следующей главе мы покажем, как получить большинство преимуществ, характерных для большой и быстрой памяти, имея только небольшую, но быструю память, хранящую лишь самую часто используемую информацию, а также медленную, но большую память, хранящую все остальное.

Упражнения

Упражнение 7.1 Предположим, что один из перечисленных ниже управляющих сигналов в однотактном процессоре RISC-V неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему? Используйте расширенную версию однотактного процессора, показанную на **рис. 7.15** и **7.16**.

(a) $RegWrite$
(b) $ALUOp_1$
(c) $ALUOp_0$
(d) $MemWrite$
(e) $ImmSrc_1$
(f) $ImmSrc_0$
(g) $ResultSrc_1$
(h) $ResultSrc_0$
(i) $PCSrc$
(j) $ALUSrc$

Упражнение 7.2 Повторите **упражнение 7.1** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 7.3 Модифицируйте однотактный процессор RISC-V так, чтобы он поддерживал одну из перечисленных ниже команд. Описание приведенных ниже команд вы можете найти в **приложении B**. Сделайте копию **рис. 7.15** и отметьте необходимые изменения в тракте данных. Обозначьте новые управляющие сигналы. Сделайте копию **табл. 7.3** и **7.6** и покажите необходимые изменения в дешифраторе АЛУ и основном дешифраторе. Также опишите изменения в схемах АЛУ, дешифратора АЛУ и основного дешифратора (**рис. 7.16**) по мере необходимости. Опишите любые другие изменения в микроархитектуре процессора, которые вы сделали.

(a) `xor`
(b) `sll`
(c) `srl`
(d) `bne`

Упражнение 7.4 Повторите **упражнение 7.3** для следующих инструкций RISC-V:

(a) `lui`
(b) `sra`
(c) `lbu`
(d) `blt`
(e) `bltu`
(f) `bge`
(g) `bgeu`
(h) `jalr`
(i) `auipc`
(j) `sb`
(k) `slli`
(l) `srai`

Упражнение 7.5 Расширьте набор команд RISC-V, включив в него `lwpostinc` (load with postincrement) – команду чтения из памяти с последующим приращением адреса. Ассемблерная инструкция `lwpostinc rd, imm(rs)` эквивалентна следующим двум инструкциям:

```
lw rd, 0(rs)
addi rs, rs, imm
```

Повторите **упражнение 7.3** для команды `lwpostinc`.

Упражнение 7.6 Расширьте набор команд RISC-V, включив в него `lwpreinc` (load with preincrement) – команду чтения из памяти с предварительным приращением адреса. Ассемблерная инструкция `lwpreinc rd, imm(rs)` эквивалентна следующим двум инструкциям:

```
lw rd, imm(rs)
addi rs, rs, imm
```

Повторите **упражнение 7.3** для команды `lwpreinc`.

Упражнение 7.7 Необходимо переделать один из блоков однотактного процессора RISC-V так, чтобы задержка этого блока уменьшилась вдвое. Используя значения задержек из **табл. 7.7**, определите, какой блок стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Какова в этом случае будет длительность такта улучшенного процессора? Объясните, почему.

Упражнение 7.8 Проанализируйте задержки в **табл. 7.7**. Бен Битдиддл разработал префиксный сумматор, уменьшающий задержку АЛУ на 20 пс. Считая, что все остальные задержки остаются неизменными, определите новую длительность такта процессора однотактного процессора RISC-V и определите, сколько времени займет выполнение тестовой программы, содержащей 100 млрд команд.

Упражнение 7.9 Измените HDL-код однотактного процессора RISC-V, приведенный в **разделе 7.6**, добавив поддержку одной из команд из **упражнения 7.3**. Доработайте тестбенч и проверочную программу (riscvtest.s и riscvtest.txt), приведенные в **разделе 7.6.3**, чтобы убедиться, что новая команда работает правильно. Все изменения сопроводите комментариями.

Упражнение 7.10 Повторите **упражнение 7.9** для новых команд из **упражнения 7.4**.

Упражнение 7.11 Предположим, что один из перечисленных ниже управляющих сигналов в многотактном процессоре RISC-V неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему? Используйте схемы тракта данных и управляющих сигналов многотактного процессора, показанные на **рис. 7.27** и **7.45**.

(a) $ResultSrc_1$
(b) $ResultSrc_0$
(c) $ALUSrcB_1$
(d) $ALUSrcB_0$
(e) $ALUSrcA_1$
(f) $ALUSrcA_0$
(g) $ImmSrc_1$
(h) $ImmSrc_0$
(i) $RegWrite$
(j) $PCUpdate$
(k) $Branch$
(l) $AdrSrc$
(m) $MemWrite$
(n) $IRWrite$

Упражнение 7.12 Повторите **упражнение 7.11** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 7.13 Модифицируйте многотактный процессор RISC-V так, чтобы он поддерживал одну из перечисленных ниже команд. Описание команд вы можете найти в приложении B.

Назовите новые управляющие сигналы. Сделайте копию **рис. 7.27** и отметьте необходимые изменения в тракте данных. Сделайте копию **рис. 7.45** и покажите все необходимые изменения в управляющем конечном автомате. Опишите любые другие необходимые изменения.

(a) `xor`
(b) `sll`
(c) `srl`
(d) `bne`

Упражнение 7.14 Повторите **упражнение 7.13** для следующих инструкций RISC-V.

(a) `lui`
(b) `sra`
(c) `lbu`
(d) `blt`
(e) `bltu`
(f) `bge`
(g) `bgeu`
(h) `jalr`
(i) `auipc`
(j) `sb`
(k) `slli`
(l) `srai`

Упражнение 7.15 Повторите **упражнение 7.5** для многотактного процессора RISC-V. Опишите все необходимые изменения в многотактном тракте данных и управляющем автомате. Возможно ли добавить эту команду без каких-либо изменений в регистровом файле? Если да, покажите, как.

Упражнение 7.16 Повторите **упражнение 7.6** для многотактного процессора RISC-V. Опишите все необходимые изменения в многотактном тракте данных и управляющем автомате. Возможно ли добавить эту команду без каких-либо изменений в регистровом файле? Если да, покажите, как.

Упражнение 7.17 Повторите **упражнение 7.7** для многотактного процессора RISC-V.

Упражнение 7.18 Повторите **упражнение 7.8** для многотактного процессора RISC-V. Используйте набор инструкций из **примера 7.7**.

Упражнение 7.19 Необходимо переделать один из блоков многотактового процессора RISC-V так, чтобы задержка этого блока существенно уменьшилась. Используя значения задержек из **табл. 7.7**, определите, какой блок стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Не надо пытаться сделать процессор быстрее, чем необходимо. Какова будет длительность такта процессора? Опишите и обоснуйте свое решение.

Упражнение 7.20 Корпорация «Голиаф» заявила, что запатентовала трехпортовый регистровый файл. Вместо того чтобы судиться с ней, Бен Битдиддл разработал новый регистровый файл, у которого всего один порт чтения/записи (как у объединенной памяти команд и данных). Переделайте тракт данных и устройство управления многотактного процессора RISC-V так, чтобы они использовали новый регистровый файл.

Упражнение 7.21 Предположим, что имеется многотактный процессор RISC-V с задержками компонентов, которые указаны в **табл. 7.7**. Также имеется новый регистровый файл, который потребляет на 40 % меньше энергии, но его задержка в два раза больше. Следует ли использовать более медленный, но экономичный регистровый файл в новой конструкции многотактного процессора? Обоснуйте свой ответ.

Упражнение 7.22 Каков CPI модифицированного многотактного процессора RISC-V из **упражнения 7.20**? Используйте набор инструкций из **примера 7.7**.

Упражнение 7.23 Сколько тактов потребуется, чтобы выполнить следующую программу на многотактном процессоре RISC-V? Чему равно CPI для этой программы?

```
   addi s0, zero, 5     # result = 5
L1:
   bge  zero, s0, Done  # если result <= 0, завершить цикл
   addi s0, s0, -1      # result = result - 1
   j L1
Done:
```

Упражнение 7.24 Повторите **упражнение 7.23** для следующей программы:

```
    addi s0, zero, 0  # i = 0
    addi s1, zero, 0  # sum = 0
```

```
    addi t3, zero, 10 # t3 = 10
Loop:
    beq  s0, t3, L2   # если i = = 10, перейти на L2
    add  s1, s1, s0   # sum = sum + i
    addi s0, s0, 1    # i = i + 1
    j Loop
L2:
```

Упражнение 7.25 Разработайте HDL-код многотактного процессора RISC-V (модуль назовите: `riscvmulti`). Он должен поддерживать инструкции, описанные в этой главе: `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `beq` и `jal`. Процессор должен быть совместим с модулем верхнего уровня, приведенным ниже. Модуль `mem` используется для хранения и команд, и данных. Вы можете воспользоваться блоками HDL-кода однотактного процессора из **раздела 7.6**. Проверьте свой процессор с помощью тестбенча и тестовой программы (riscvtest.s и riscvtest.txt) из **раздела 7.6.3**. Опишите все изменения кода.

```
module top(input logic clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic MemWrite);
   logic [31:0] ReadData;
   // объявление процессора и памяти
   riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                      WriteData, ReadData);
   mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

Упражнение 7.26 Добавьте в HDL-код многотактного процессора RISC-V из **упражнения 7.25** поддержку одной из новых команд из **упражнения 7.14**. Доработайте тестбенч и проверочную программу (riscvtest.s и riscvtest.txt) из **раздела 7.6.3**, чтобы убедиться, что новая команда работает правильно. Опишите все изменения кода.

Упражнение 7.27 Повторите **упражнение 7.26** для одной из новых команд из **упражнения 7.13**.

Упражнение 7.28 Конвейерный процессор RISC-V выполняет приведенную ниже программу. Какие регистры он читает и в какие регистры пишет на пятом такте? Напомним, что у конвейерного процессора RISC-V есть блок предотвращения конфликтов. Предположим, что блок памяти возвращает результат в течение одного цикла.

```
addi s1, zero, 11 # s1 = 11
lw   s2, 25(s0)   # s2 = memory[s0+25]
add  s3, s3, s4   # s3 = s3 + s4
or   s4, s1, s2   # s4 = s1 | s2
lw   s5, 16(s2)   # s5 = memory[s2+16]
```

Упражнение 7.29 Повторите **упражнение 7.28** для следующего фрагмента кода RISC-V:

```
xor  s1, s2, s3 # s1 = s2 ^ s3
addi s0, s3, -4 # s0 = s3 - 4
lw   s3, 16(s7) # s3 = memory[s7+16]
```

```
sw   s4, 20(s1) # memory[s1+20] = s4
or   t2, s0, s1 # t2 = s0 | s1
```

Упражнение 7.30 Повторите **упражнение 7.28** для следующего фрагмента кода RISC-V:

```
addi s1, zero, 11 # s1 = 11
lw   s2, 25(s1)   # s2 = memory[36]
lw   s5, 16(s2)   # s5 = memory[s2+16]
add s3, s2, s5   # s3 = s2 + s5
or  s4, s3, t4   # s4 = s3 | t4
and s2, s3, s4   # s2 = s3 & s4
```

Упражнение 7.31 Повторите **упражнение 7.28** для следующего фрагмента кода RISC-V:

```
addi s1, zero, 52 # s1 = 52
addi s0, s1, -4   # s0 = s1 - 4 = 48
lw   s3, 16(s0)   # s3 = memory[64]
sw   s3, 20(s0)   # memory[68] = s3
xor  s2, s0, s3   # s2 = s0 ^ s3
or   s2, s2, s3   # s2 = s2 | s3
```

Упражнение 7.32 Используя такую же диаграмму, как на **рис. 7.57**, покажите пересылки через байпас и приостановки конвейера, возникающие при выполнении инструкций из **упражнения 7.30** на конвейерном процессоре RISC-V.

Упражнение 7.33 Повторите **упражнение 7.32** для инструкций из **упражнения 7.31**.

Упражнение 7.34 Сколько тактов потребуется конвейерному процессору RISC-V для выполнения всех инструкций программы из **упражнения 7.30**? Чему равно CPI процессора для этой программы?

Упражнение 7.35 Повторите **упражнение 7.34** для программы из **упражнения 7.31**.

Упражнение 7.36 Объясните, как добавить в конвейерный процессор RISC-V поддержку команды непосредственной записи в старшие разряды `lui`. Назовите новые управляющие сигналы. Скопируйте **рис. 7.61** и покажите на нем изменения в тракте данных. Скопируйте **табл. 7.3** и **7.6** и покажите в них изменения в дешифраторе АЛУ и основном дешифраторе. Если нужны другие изменения, опишите их тоже.

Упражнение 7.37 Повторите **упражнение 7.36** для инструкции `xor`.

Упражнение 7.38 Производительность конвейерного процессора могла бы быть выше, если бы вычисление условия перехода выполнялось бы на стадии *Decode*, а не на стадии *Execute*. Объясните, как модифицировать конвейерный процессор, показанный на **рис. 7.61**, чтобы вычисление условия перехода выполнялось на стадии *Decode*. Как изменятся сигналы приостановки, очистки конвейера и пересылки? Вычислите новое значение CPI, длительность такта процессора и общее время выполнения программы в **примерах 7.9** и **7.10**.

Упражнение 7.39 Необходимо переделать один из блоков конвейерного процессора RISC-V так, чтобы задержка этого блока существенно уменьшилась.

Используя значения задержек из **табл. 7.7** (задержка компаратора равенства составляет 23 пс), определите, какой блок стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Не надо пытаться сделать процессор быстрее, чем необходимо. Какова будет длительность такта процессора? Опишите и объясните свое решение.

Упражнение 7.40 Проанализируйте задержки в **табл. 7.7**. Предположим, что АЛУ теперь работает на 20 % быстрее. Изменится ли длительность такта конвейерного процессора RISC-V? Что произойдет, если АЛУ станет на 20 % медленнее? Обоснуйте ответ.

Упражнение 7.41 Предположим, что конвейерный процессор RISC-V поделен на 10 стадий длительностью 400 пс каждая, включая все накладные расходы на конвейеризацию. Пусть процентное соотношение разных типов команд такое же, как в **примере 7.7**. При этом в половине случаев результат команд чтения из памяти требуется немедленно, что приводит к приостановке конвейера на шесть тактов. Также будем считать, что 30 % условных переходов предсказываются неверно, а адрес перехода для команд условного и безусловного переходов вычисляется в конце второй стадии. Расчитайте для этого десятистадийного процессора среднее значение CPI и время выполнения 100 млрд команд из бенчмарка SPECINT2000.

Упражнение 7.42 Разработайте HDL-код конвейерного процессора RISC-V и назовите модуль riscv. Процессор должен быть совместим с модулем верхнего уровня, указанным ниже. Он должен поддерживать все инструкции, рассмотренные в этой главе: `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `beq` и `jal`. Вы можете воспользоваться блоками HDL-кода однотактного процессора из **раздела 7.6**. Проверьте свой процессор с помощью тестбенча и тестовой программы (riscvtest.s и riscvtest.txt) из **раздела 7.6.3**, предварительно модифицировав тестбенч. Опишите все изменения кода.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // определение процессора и памяти
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
              WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule
```

Упражнение 7.43 Модифицируйте HDL-код конвейерного процессора RISC-V из **упражнения 7.42** для поддержки инструкции `xor` из **упражнения 7.37**. Доработайте тестбенч и проверочную программу (riscvtest.s и riscvtest.txt) из **раздела 7.6.3**, чтобы убедиться, что новая команда работает правильно.

Упражнение 7.44 Модифицируйте HDL-код конвейерного процессора RISC-V из **упражнения 7.42** для поддержки инструкции `lui` из **упражнения 7.36**. Доработайте тестбенч и проверочную программу (riscvtest.s и riscvtest.txt) из **раздела 7.6.3**, чтобы убедиться, что новая команда работает правильно.

Упражнение 7.45 Разработайте для конвейерного процессора RISC-V блок разрешения конфликтов, приведенный на **рис. 7.61**. Используйте HDL-код для описания вашего модуля. Изобразите в общих чертах схему, которую мог бы сгенерировать из вашего кода HDL-синтезатор.

Упражнение 7.46 Покажите, как изменить многотактный процессор RISC-V, чтобы он выдавал исключение, если встречается недопустимая инструкция. Мы рассмотрели исключения в **разделе 6.6.2**. Код причины исключения для неопределенной команды – 2 (**табл. 6.6**). При необходимости доработайте как тракт данных (**рис. 7.27**), так и блок управления, включая главный конечный автомат (**рис. 7.45**). Предположим, что `mtvec` уже готов и содержит адрес обработчика исключений.

Упражнение 7.47 Повторите **упражнение 7.46** для исключения по событию неверного адреса чтения из памяти, код причины которого равен 4 (**табл. 6.6**).

Упражнение 7.48 Покажите, как модифицировать многотактный процессор RISC-V для реализации привилегированной инструкции `csrrw` (чтение/запись CSR). На **рис. 7.76** показан машинный код для инструкции ассемблера `csrrw x9, mscratch, x8`, которая одновременно копирует значение `mscratch` в регистр `x9` и значение регистра `x8` в `mscratch`. Число CSR кодируется в 12-битном непосредственном поле инструкции типа *I*. Здесь `mscratch` – это число CSR 0x340. Любое 12-битное число CSR должно быть доступно для чтения/записи. В **табл. B.8** в **приложении B** приведена дополнительная информация о привилегированных форматах команд. При необходимости доработайте как канал данных (**рис. 7.27**), так и блок управления, включая главный конечный автомат (**рис. 7.45**), для поддержки команды `csrrw`.

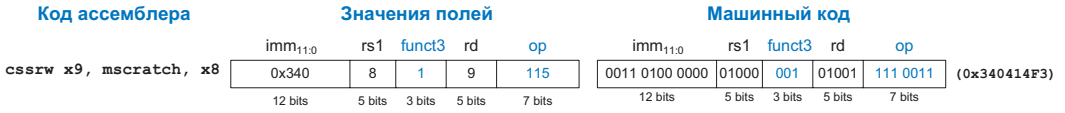


Рис. 7.76 Привилегированная инструкция `csrrw` (чтение/запись CSR)

Упражнение 7.49 Повторите **упражнение 7.48** для привилегированной инструкции `csrrs` (чтение и установка CSR). На **рис. 7.77** показан машинный код для инструкции ассемблера `csrrs x7, mcause, x3`, который одновременно копирует значение mcause в x7 и записывает результат операции `mcause|x3` в `mcause`. Здесь `mscratch` – это число CSR 0x342. Любое 12-битное число CSR должно быть доступно для чтения/установки.

Рис. 7.77 Привилегированные инструкции `csrrs` (чтение/установка CSR)

Вопросы для собеседования

Ниже приведены некоторые вопросы, которые задают на собеседованиях на вакансии разработчиков цифровых устройств.

Вопрос 7.1 Объясните преимущества конвейерных микропроцессоров.

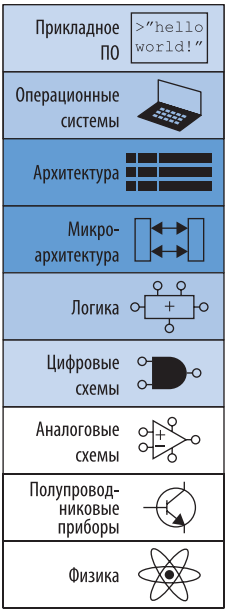
Вопрос 7.2 Если большее количество стадий конвейера позволяет процессору работать быстрее, почему нет процессоров с сотней стадий?

Вопрос 7.3 Объясните, что такое конфликты в микропроцессоре и каковы пути их разрешения. Какие преимущества и недостатки у каждого из способов?

Вопрос 7.4 Расскажите, что такое суперскалярный процессор, каковы его достоинства и недостатки.

ГЛАВА 8

Системы памяти



8.1. Введение

Производительность компьютерной системы зависит от системы памяти, а также от микроархитектуры процессора. В **главе 7** мы предполагали, что компьютер оснащен идеальной системой памяти, доступ к которой можно получить за один такт. Но это будет верно только для очень маленького объема памяти или очень медленного процессора! Первые процессоры были относительно медленными, поэтому память не отставала. Но скорость работы процессора увеличивалась быстрее, чем скорость памяти. В настоящее время оперативная память типа DRAM (Dynamic Random Access Memory, динамическая память с произвольным доступом) медленнее процессора от 10 до 100 раз. Нарастающий разрыв в скорости между процессором и оперативной памятью ведет к использованию все более сложных подсистем памяти в стремлении приблизить скорость работы памяти к скорости процессора. Первая половина этой главы рассказывает о подсистемах памяти и анализирует различные компромиссы между их скоростью, объемом и ценой.

Процессор работает с памятью через интерфейс памяти (memory interface). На **рис. 8.1** показан простой интерфейс памяти, используемый

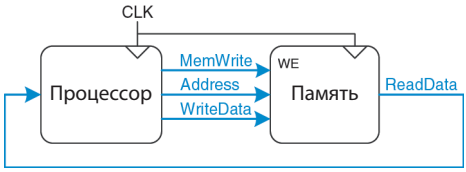


Рис. 8.1 Интерфейс памяти

в нашем многотактном процессоре RISC-V. Процессор выставляет адреса на шину адреса (*Address*), идущую к подсистеме памяти. Для чтения управляющий сигнал записи (*MemWrite*) устанавливают в низкий уровень, а память возвращает данные по *шине чтения данных* (*ReadData*). Для записи *MemWrite* устанавливается в единицу, и процессор посылает данные в память по *шине записи данных* (*WriteData*).

Основные проблемы при разработке подсистемы памяти можно объяснить, рассматривая в качестве метафоры книги в библиотеке. В библиотеке на полках есть много книг. Если вы собрались написать научную работу по толкованию снов, вы можете пойти в библиотеку[1], взять с полки книгу Зигмунда Фрейда «Толкование сновидений» и принести ее к себе в кабинет. После того как вы ее просмотрите, вы можете вернуть ее обратно и взять работу Карла Густава Юнга «Психология подсознания». Затем вы могли бы пойти обратно за «Толкованием сновидений», чтобы использовать еще одну цитату из нее. А потом опять пойти в библиотеку за книгой Зигмунда Фрейда «Я и Оно». Очень скоро вы устанете бегать в библиотеку, и если вы достаточно сообразительны, то будете просто держать нужные книги в своем кабинете, вместо того чтобы бегать за ними туда-обратно. Более того, когда вы возьмете книгу Зигмунда Фрейда, то можете взять еще несколько его книг с той же полки (на всякий случай).

Эта метафора иллюстрирует принцип, изложенный в **разделе 6.2.1**, – «типичный сценарий должен быть быстрым». Оставляя в своем кабинете книги, которые вы только что использовали или которые вы, может быть, будете скоро использовать, вы уменьшаете количество отнимающих много времени походов в библиотеку. В частности, вы используете принципы *временно́й* (temporal) и *пространственной* (spatial) *локальности*. Временная локальность означает, что если вы только что использовали книгу, то, вероятно, она вам снова скоро понадобится. Пространственная локальность означает, что когда вам понадобилась определенная книга, то, вероятно, вас заинтересуют и другие книги с той же полки.

Библиотека сама старается оптимизировать типичный сценарий, используя принцип локальности. У нее нет ни места на полках, ни денег, чтобы собрать все книги в мире. Вместо этого она хранит редко используемые книги в подвале. Также она использует систему межбиблиотечного

[1] Мы понимаем, что в век интернета использование библиотек среди учащихся стремительно падает. Но мы верим, что в библиотеках хранятся огромные богатства знаний, которые достались человечеству тяжелым трудом, и не все они доступны в электронном виде. Мы надеемся, что искусство поиска знаний в книгах не будет полностью вытеснено запросами во Всемирной паутине.

обмена с соседними библиотеками, так что она может предложить вам больше книг, чем физически имеется в наличии.

В итоге вы получаете выгоду как от большой коллекции книг, так и от быстрого доступа к наиболее популярным книгам, применяя иерархию хранения книг. Наиболее часто используемые книги находятся у вас на столе. Более многочисленная коллекция – на полках вашей библиотеки. Еще большая коллекция книг доступна из хранилища и из других библиотек. Точно так же подсистемы памяти используют иерархию хранилищ для быстрого доступа к наиболее часто используемым данным, одновременно обеспечивая возможность хранения больших объемов данных.

Подсистемы памяти, применяемые для создания такой иерархии, были описаны в **разделе 5.5**. Компьютерная память в основном построена на базе динамической (DRAM) и статической (SRAM) памяти. В идеале память должна быть быстрой, большой и дешевой. Но на практике любой тип памяти имеет только два из этих свойств; память либо медленная, либо дорогая, либо маленького объема. Несмотря на это, компьютерные системы могут приближаться к идеалу, сочетая дешевую, быструю и маленькую память с дешевой, медленной и большой. Быстрая память используется для хранения часто используемых данных и команд, так что создается впечатление, что подсистема памяти всегда работает довольно быстро. Остальные данные и команды хранятся в большой памяти, которая работает медленнее, но позволяет иметь большой общий объем памяти. Комбинация двух дешевых типов памяти – это намного менее дорогой вариант, чем одна большая и быстрая память. Этот принцип распространяется на всю иерархию памяти, так как с увеличением объема памяти уменьшается ее скорость работы.

Напомним, что быстродействие системы определяется как задержкой, так и пропускной способностью. *Задержка памяти* – это время доступа к первому байту информации. *Пропускная способность* – это количество байтов, которое можно извлечь из памяти за секунду. Многие системы памяти имеют хорошую пропускную способность, но большие задержки.

Оперативная память компьютера обычно строится на микросхемах динамической памяти (DRAM). В 2021 году типичный персональный компьютер имел оперативную память DRAM размером 8–32 Гб, и она стоила около трех долларов за гигабайт. Цены на DRAM падали в среднем на 15–25 % в год на протяжении последних трех десятилетий, при этом емкость памяти росла примерно с такой же скоростью, так что общая цена памяти в персональном компьютере оставалась приблизительно одинаковой. К сожалению, скорость работы самих микросхем DRAM

возрастала только на 7 % в год, в то время как производительность процессоров возрастала на 25–50 % в год. На **рис. 8.2** показан график увеличения скорости работы оперативной памяти и процессоров с 1980 года по настоящее время. В начале 1980-х годов скорость процессоров и памяти была примерно одинаковой, но затем разрыв в производительности сильно увеличился и память серьезно отстала[1].

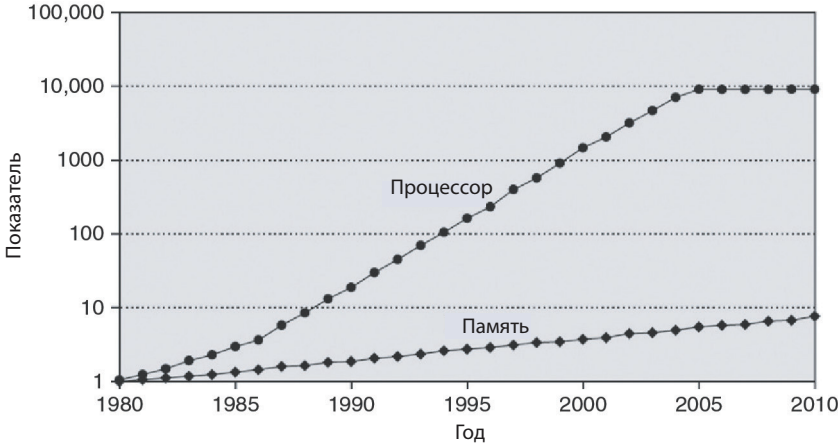


Рис. 8.2 Разница в производительности процессора и памяти (график из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2011, с разрешения авторов)

Память DRAM могла успешно идти в ногу с процессорами в 1970-х и в начале 1980-х годов, но сейчас она, к сожалению, слишком медленная. Время доступа к данным в DRAM на порядок или два медленнее, чем длительность такта процессора (десятки наносекунд против долей наносекунды). При этом пропускная способность DRAM весьма высока – порядка 30 ГБ/с.

Чтобы справиться с этой проблемой, компьютеры хранят наиболее часто используемые команды и данные в быстрой, но небольшой по объему памяти, называемой кеш-памятью, или просто *кешем* (cache). Кеш-память обычно сделана с использованием статической памяти (SRAM) и находится в той же микросхеме, что и процессор. Скорость кеша сравнима со скоростью процессора, так как, во-первых, память SRAM работает быстрее, чем DRAM, а во-вторых, она находится в чипе процессора и позволяет избавиться от задержек распространения сигналов по пути к внешним микросхемам памяти.

[1] Хотя производительность отдельных ядер в процессорах остается примерно одинаковой где-то с 2005 года, как показано на рис. 8.2, но переход на многоядерные системы (на рисунке не показан) только усугубляет разрыв в производительности процессоров и памяти.

В 2021 году стоимость SRAM, расположенной в чипе процессора, составляла порядка 100 долларов за Гбайт, но так как размер кеша сравнительно мал (от нескольких Кбайт до нескольких Мбайт), то общая стоимость кеша не такая большая. Кеш-память может хранить как данные, так и команды, но для краткости мы будем говорить, что она содержит просто «данные». Задержка SRAM варьируется от нескольких десятых наносекунды для кеш-памяти размером 16 Кбайт до нескольких наносекунд для кеш-памяти размером 4 Мбайта. Пропускная способность может достигать сотен ГБ/с.

Если процессор запрашивает данные, которые уже находятся в кеше, то он получает их очень быстро. Это называется *попаданием в кеш* (cache hit). В противном случае процессор вынужден читать данные из оперативной памяти (DRAM). Это называется *промахом кеша*, кеш-промахом или промахом доступа в кеш (cache miss). Если процессор попадает в кеш бóльшую часть времени, то он редко простаивает в ожидании доступа к медленной оперативной памяти, и среднее время доступа мало.

Третий уровень в иерархии памяти – *жесткий диск*. Аналогично тому, как библиотека использует подвал для хранения книг, не умещающихся на полках, компьютерные системы применяют жесткий диск для хранения данных, которые не помещаются в оперативной памяти. В 2021 году жесткий диск, сделанный на основе технологии магнитной записи (Hard Disk Drive, HDD), стоил менее 0,03 доллара США за Гбайт и имел время доступа от 5 до 10 мс. Пропускная способность такого диска составляет порядка 100 МБ/с для больших файлов и до 1 МБ/с для произвольного доступа к небольшим (4 Кбайт) файлам. Цены на жесткие диски снижаются на 60 % в год, но время доступа почти не улучшается. *Твердотельные диски* (Solid State Drive, SSD), которые сделаны на основе флеш-памяти, становятся все более популярной альтернативой HDD. SSD использовались для специальных применений на протяжении двух десятилетий и вышли на потребительский рынок только в 2007 году. Они не подвержены механическим отказам, но и стоят в 3–4 раза дороже, чем HDD, – 0,10 доллара за Гбайт. С появлением на рынке твердотельных накопителей разница в цене между ними и жесткими дисками сократилась, и соответственно возросла популярность твердотельных накопителей. SSD-диски имеют время доступа менее 0,1 мс. Пропускная способность может составлять от 500 до 3000 МБ/с для больших файлов и от 50 до 250 МБ/с для файлов размером 4 Кбайт.

Жесткий диск создает иллюзию наличия бóльшего объема памяти, чем реально доступно в оперативной памяти. Это называется виртуальной памятью. Как и доступ к книгам в хранилище, доступ к данным в виртуальной памяти занимает длительное время. Оперативная память, также называемая физической памятью, содержит только часть данных, находящихся в виртуальной памяти, остальные данные находятся на жестком диске. Следовательно, оперативную память можно рассматри-

вать как кеш-память для наиболее часто используемых данных с жесткого диска.

В оставшейся части этой главы мы рассмотрим иерархию памяти компьютерной системы, показанную на **рис. 8.3**. Процессор сначала ищет данные в маленькой, но быстрой кеш-памяти, обычно расположенной на той же самой микросхеме. Если данные в кеше отсутствуют, процессор обращается к оперативной памяти. Если данных нет и там, то процессор читает данные из большого, хотя и медленного, жесткого диска, используя механизм виртуальной памяти. **Рисунок 8.4** иллюстрирует соотношение емкости и скорости в многоуровневой иерархии памяти компьютерной системы и показывает типичную стоимость, время доступа и пропускную способность для технологий памяти по состоянию на 2021 год. Как видите, с уменьшением времени доступа скорость возрастает.

В **разделе 8.2** мы покажем, как анализировать производительность систем памяти. В **разделе 8.3** мы рассмотрим несколько методов организации кеш-памяти, а в **разделе 8.4** расскажем о виртуальной памяти.

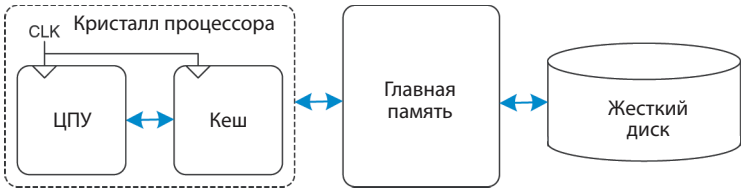
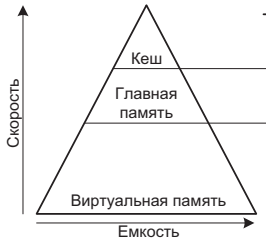


Рис. 8.3 Типичная иерархия памяти



Технология	Цена за ГБ	Время доступа (нс)	Пропускная способность (ГБ/с)
SRAM	$100	0,2–3	100+
DRAM	$3	10–50	30
SSD	$0,10	20 000	0,05–3
HDD	$0,03	5 000 000	0,001–0,1

Рис. 8.4 Компоненты иерархии памяти и их характеристики на 2021 год

8.2. Анализ производительности систем памяти

Чтобы оценить соотношение цены и производительности у разных вариантов систем памяти, разработчикам (и покупателям) компьютеров нужны количественные способы измерения производительности. Мерами измерения производительности систем памяти являются доля попаданий (hit rate) или промахов (miss rate), а также среднее время доступа.

Доля попаданий и промахов вычисляется так:

$$\textit{Доля промахов} = \frac{\textit{Количество промахов}}{\textit{Общее количество доступов к памяти}} = 1 - \textit{Доля попаданий};$$
$$\textit{Доля попаданий} = \frac{\textit{Количество попаданий}}{\textit{Общее количество доступов к памяти}} = 1 - \textit{Доля промахов}. \quad (8.1)$$

Пример 8.1 ВЫЧИСЛЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ КЕШ-ПАМЯТИ

Предположим, что программа содержит 2000 команд обращения к данным (чтения и записи), но только 1250 из этих команд нашли запрошенные ими данные в кеш-памяти. Остальным 750 командам пришлось получать данные из оперативной памяти или с жесткого диска. Чему равны доли промахов и попаданий в кеш-память в этом случае?

Решение Доля промахов вычисляется как 750 / 2000 = 0,375 = 37,5 %. Доля попаданий равна 1250 / 2000 = 0,625 = 1 − 0,375 = 62,5 %.

Среднее время доступа к памяти (average memory access time, AMAT) – это среднее время, которое процессор тратит, ожидая доступа к памяти при выполнении команд загрузки или сохранения данных. В типичной компьютерной системе, показанной на **рис. 8.3**, процессор сначала ищет данные в кеше. Если данных в кеше нет, то процессор обращается к оперативной памяти. Если же данных нет и там, то процессор выполняет обращение к виртуальной памяти на диске. Следовательно, AMAT вычисляется так:

$$\text{AMAT} = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM}), \quad (8.2)$$

где t_{cache}, t_{MM}, и t_{VM} – это времена доступа к кешу, оперативной памяти и диску соответственно, а MR_{cache} и MR_{MM} – это доли промахов кеша и оперативной памяти.

Пример 8.2 ВЫЧИСЛЕНИЕ СРЕДНЕГО ВРЕМЕНИ ДОСТУПА К ПАМЯТИ

Предположим, что компьютерная система имеет память всего с двумя уровнями иерархии: кешем и оперативной памятью. Чему равно среднее время доступа? Время доступа и процент промахов заданы в табл. 8.1.

Решение: среднее время доступа к памяти составляет 1 + 0,1(100) = 11 тактов.

Таблица 8.1 Время доступа и доля промахов

Уровень памяти	Время доступа в тактах	Доля промахов
Кеш-память	1	10 %
Оперативная память	100	0 %

Пример 8.3 УЛУЧШЕНИЕ ВРЕМЕНИ ДОСТУПА

Среднее время доступа к памяти в 11 тактов означает, что процессор тратит 10 тактов на ожидание данных на каждый такт реального использования этих данных. Какой процент промахов в кеш необходим для уменьшения среднего времени доступа к памяти до 1,5 такта при заданном в **табл. 8.1** времени доступа к памяти?

Решение Обозначим процент промахов кеша как m, тогда среднее время доступа будет равно $1 + 100m$. Вычислим m, приравняв это выражение к 1,5. Ответ: требуемый процент промахов должен быть равен 0,5 %.

Джин Амдал, 1922–2015

Джин Амдал (Gene Amdahl) наиболее известен как автор «закона Амдала» – наблюдения, которое он сделал в 1965 году. Будучи аспирантом, Амдал начал в свободное время разрабатывать компьютеры. Эта работа принесла ему степень доктора философии по теоретической физике (Ph. D., западный аналог степени кандидата наук) в 1952 году. Сразу после окончания аспирантуры Амдал устроился на работу в IBM, а позже основал три компании, одну из которых в 1970 году назвал Amdahl Corporation.

Мы должны предупредить: улучшение производительности в реальности может быть не таким привлекательным, как оно выглядит на бумаге. Например, увеличение скорости памяти в десять раз не обязательно сделает компьютерную программу в десять раз быстрее. Если 50 % команд в программе – это команды загрузки и сохранения данных, то десятикратное увеличение скорости памяти приведет к ускорению программы всего лишь в 1,82 раза. Этот общий принцип называется *законом Амдала* и гласит, что усилия, потраченные на улучшение производительности подсистемы, оправдываются только тогда, когда она оказывает значительное влияние на общую производительность системы.

8.3. Кеш-память

Кеш содержит часто используемые данные из памяти. Количество слов данных, которое он может хранить, называется *емкостью* (capacity) кеша. Поскольку емкость кеша меньше, чем емкость оперативной памяти, то разработчик компьютерной системы должен решить, какое подмножество оперативной памяти хранить в кеше.

Кеш – потайное место для хранения оружия и продовольствия (словарь Merriam Webster Online Dictionary, 2012, www.merriam-webster.com).

Когда процессор пытается получить доступ к данным, он сначала ищет их в кеше. Если данные там есть, то есть произошло попадание в кеш, то процессор получает их немедленно. Если же их там нет, то есть произошел промах кеша, то процессор извлекает данные из оперативной памяти и помещает их в кеш для последующего использования. Для этого кеш должен заменить какие-то старые данные на новые. В этом разделе мы рассмотрим разработку кешей, попытавшись ответить на следующие вопросы: (1) Какие данные хранятся в кеш-памяти? (2) Как найти данные в кеш-памяти? и (3) Какие данные заместить в кеш-памяти, когда нужно разместить новые данные, а кеш заполнен?

При чтении следующих разделов помните, что ответы на эти вопросы связаны с присущей большинству программ пространственной и временно́й локальностью при обращении к данным. Эту локальность кеш использует для предсказания того, какие данные понадобятся следующими. Если программа обращается к памяти в случайном порядке, она не получит никакой выгоды от использования кеша[1].

Как мы увидим в следующих разделах, кеш-память характеризуется емкостью C (capacity), количеством наборов S (set), длиной строки, иногда называемой размером блока b (block), количеством строк или блоков B и степенью ассоциативности N.

Хотя мы сфокусируем внимание на чтении из кеша данных, но те же самые принципы применяются и для чтения из кеша команд. Запись в кеш данных похожа на чтение и будет рассмотрена в **разделе 8.3.4**[2].

8.3.1. Какие данные хранятся в кеш-памяти?

Идеальный кеш должен предугадывать, какие данные понадобятся процессору, и выбирать их из оперативной памяти заранее таким образом, чтобы кеш имел нулевой процент промахов. Но поскольку точно предсказать будущее невозможно, то кеш должен угадывать, какие данные понадобятся, основываясь на предыдущих обращениях в память. В частности, кеш использует временну́ю и пространственную локальность, чтобы уменьшить процент промахов в кеш.

Напомним, что временна́я локальность означает, что процессор, вероятно, еще раз обратится к тем данным, которые он недавно использовал. Поэтому, когда процессор читает или записывает данные, которых нет в кеше, то эти данные копируются из оперативной памяти в кеш, так что последующие обращения к ним уже не вызовут промаха кеша.

Напомним также, что пространственная локальность означает, что когда процессор обращается к каким-либо данным, то, вероятно, ему понадобятся и расположенные рядом данные.

Поэтому когда кеш читает одно слово данных из памяти, он заодно читает и несколько соседних слов. Эта группа слов называется *строкой кеша* (cache line), также иногда используют термин «*блок кеша*» (cache block). Количество слов в строке b называется *длиной строки* (line size или block size). Кеш емкостью C содержит $B = C / b$ строк.

Принципы пространственной и временно́й локальности данных были экспериментально подтверждены на реальных программах. Если переменная используется в программе, то та же самая переменная будет, скорее всего, использована снова, тем самым создавая временную локаль-

---

[1] Скорее всего, такая программа будет работать даже медленнее, так как кеш-памяти присущи определенные накладные расходы. – *Прим. перев.*

[2] Во многих архитектурах запись в кеш команд не имеет смысла и потому не реализована. – *Прим. перев.*

ность. Если используется какой-либо элемент массива, то, скорее всего, и другие элементы этого массива тоже будут использованы, тем самым создавая пространственную локальность.

8.3.2. Как найти данные в кеш-памяти?

Кеш состоит из S наборов, каждый из которых содержит одну или несколько строк (блоков данных). Взаимосвязь между адресом данных в оперативной памяти и расположением этих данных в кеше называется *отображением*. Каждый адрес памяти всегда отображается в один и тот же набор кеша. Несколько битов адреса используются, чтобы определить, какой именно набор кеша содержит искомые данные. Если в наборе больше одной строки, то данные могут находиться в любой из них.

Кеш-память классифицируется по количеству строк в наборе. В *кеше прямого отображения* (direct mapped cache) каждый набор содержит только одну строку (один блок), так что кеш содержит $S = B$ наборов. Таким образом, каждый из адресов в оперативной памяти отображается в одну-единственную строку кеша. В случае же наборно-ассоциативного кеша с N секциями (N-way set associative cache) каждый набор состоит из N строк. Каждый адрес памяти по-прежнему отображается в один-единственный набор, но количество наборов в этом случае равно $S = B/N$, а данные могут оказаться в любой из N строк этого набора. В отличие от кеша прямого отображения и наборно-ассоциативного кеша, *полностью ассоциативный кеш* (fully associative cache) имеет только один набор ($S = 1$), и данные могут оказаться в любой из B строк этого набора. Таким образом, полностью ассоциативный кеш – это то же самое, что и наборно-ассоциативный кеш с B секциями (количество секций совпадает с количеством строк во всем кеше).

Для иллюстрации этих вариантов организации кеша мы рассмотрим подсистему памяти процессора RISC-V с 32-битными адресами и 32-битными словами. В наших примерах память адресуется побайтно, а каждое слово состоит из четырех байт, так что память содержит 2^{30} слов, выровненных по 4-байтной границе (т. е. находящихся по адресам 0, 4, 8, ...). Для простоты мы будем рассматривать кеши с емкостью $C = 8$ слов. Мы начнем с длины строки (b), равной одному слову, после чего перейдем к более длинным строкам.

Кеш-память прямого отображения

В кеш-памяти прямого отображения каждый набор содержит только одну строку (блок данных), так что у него $S = B$ наборов и строк. Чтобы понять способ отображения адресов памяти в определенные строки такого кеша, представьте, что оперативная память поделена на блоки по b слов так же, как кеш поделен на строки по b слов. Адрес одного из слов, находящихся в блоке 0 оперативной памяти, отображается в на-

бор 0 кеша. Адрес слова из блока 1 оперативной памяти отображается в набор 1 кеша, и так далее, пока адрес слова из блока $B - 1$ оперативной памяти не отобразится в строку $B - 1$ кеша. Больше строк в кеше нет, так что следующий блок оперативной памяти (блок B) снова отображается в строку 0 кеша, и так далее.

Отображение для такого кеша емкостью 8 слов и размером строки в одно слово показано на **рис. 8.5**. В кеше 8 наборов, каждый из них содержит по одной строке, длина которых равна одному слову. Младшие два бита адреса всегда равны нулю, потому что все адреса выровнены на границу слова. Следующие $\log_2 8 = 3$ бита адресуют один из восьми наборов, в который будет отображен этот адрес памяти. Таким образом, данные из адресов 0x00000004, 0x00000024, ..., 0xFFFFFFE4 будут отображены в один и тот же набор 1, как это показано синим цветом. Аналогично данные из адресов 0x00000010, ..., 0xFFFFFFF0 отображаются в набор 4, и так далее. Каждый адрес оперативной памяти отображается строго в один набор кеша.

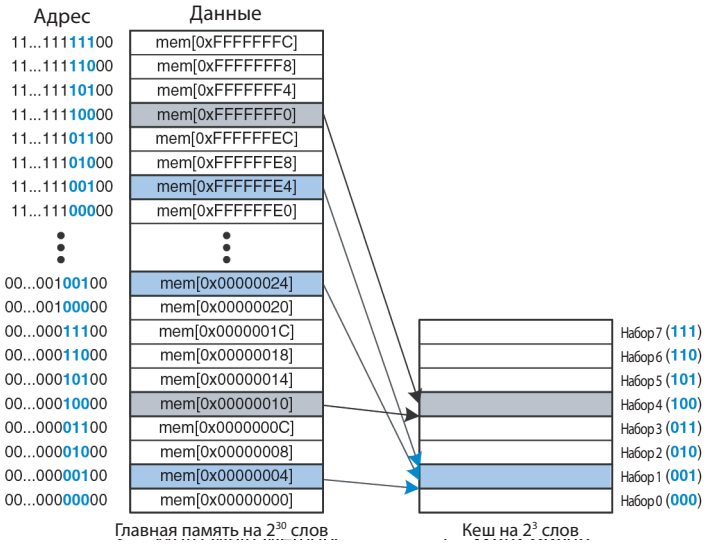


Рис. 8.5 Отображение оперативной памяти на кеш прямого отображения

Пример 8.4 ЧАСТИ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЕШ

В какой набор кеша на **рис. 8.5** будет отображено слово с адресом 0x00000014? Назовите другой адрес, который отображается в этот же самый набор.

Решение Два младших бита адреса всегда равны нулю, потому что адрес выровнен по границе слова. Следующие 3 бита равны 101, так что слово будет отображено в набор 5. Слова с адресами 0x34, 0x54, 0x74, ..., 0xFFFFFFF4 тоже будут отображены в этот набор.

Так как в один набор кеша отображается множество адресов, то кеш должен отслеживать адреса данных, находящихся в каждом из наборов в текущий момент времени. Младшие биты адреса определяют набор, в котором хранятся данные. Оставшиеся биты адреса называются *тегом* (tag) и указывают, какой именно из всех возможных адресов сейчас находится в этом наборе.

В наших предыдущих примерах два младших бита адреса называются *байтовым смещением* (byte offset), поскольку они указывают на номер байта внутри слова. Следующие три бита называются *индексом* (cache index) или номером набора (set bits), так как они указывают на *номер набора*, в который отображается этот адрес (в общем случае номер набора состоит из $\log_2 S$ бит, где S – количество наборов). Оставшиеся 27 бит тега указывают на адрес слова, которое в текущий момент находится в этом наборе кеша. На **рис. 8.6** показано, на какие части делится адрес 0xFFFFFFE4. Он отображается в набор 1, а его тег содержит одни единицы.

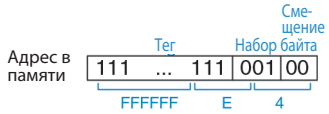


Рис. 8.6 Части адреса 0xFFFFFFE4 при отображении в кеш на рис. 8.5

Пример 8.5 ЧАСТИ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЕШ

Найти количество битов тега и номера набора (индекса) для кеш-памяти прямого отображения с 1024 (2^{10}) наборами и длиной строки, равной одному слову. Размер адреса равен 32 битам.

Решение Для кеш-памяти, у которой 2^{10} наборов, требуется $\log_2(2^{10}) = 10$ бит для хранения номера набора (индекса). Два младших бита адреса хранят байтовое смещение, а оставшиеся $32 - 10 - 2 = 20$ бит используются для тега.

Иногда, особенно когда компьютер только включили, наборы кеша еще не содержат никаких данных. Для каждого набора в кеше есть *бит достоверности* (valid bit), который равен единице, если в нем находятся корректные данные, и нулю, если находящееся в нем значение случайное.

На **рис. 8.7** изображена блок-схема аппаратной реализации кеша прямого отображения, показанного на **рис. 8.5**. Кеш использует блок статической памяти SRAM с восемью ячейками. Каждая ячейка, или набор (Set), содержит 32 бита данных (Data), 27 бит тега (Tag) и 1 бит достоверности (V). К кешу обращаются, используя 32-битные адреса. Два младших бита адреса – байтовое смещение – игнорируются при обращении к словам. Следующие 3 бита указывают на ячейку или набор кеш-памяти. Команда загрузки читает эту ячейку из кеш-памяти и проверяет тег и бит достоверности. Если тег совпадает со старшими 27 битами адреса и бит достоверности равен 1, то происходит попадание в кеш (Hit) и данные передаются процессору. В противном случае происходит промах кеша и подсистема памяти должна прочитать запрошенные данные из оперативной памяти.

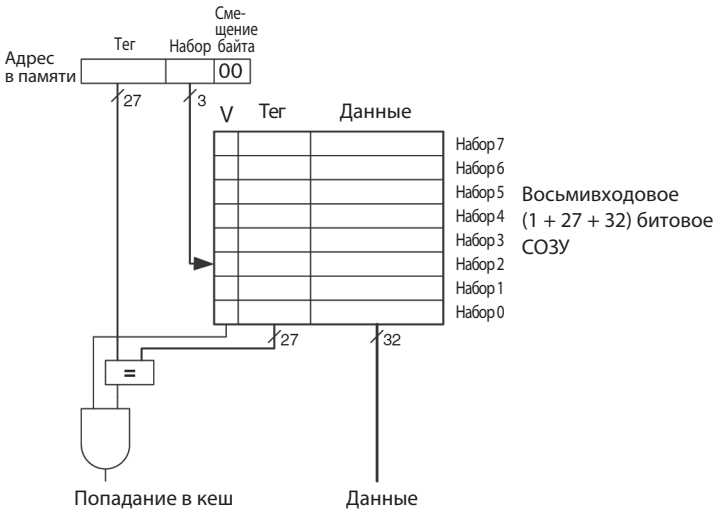


Рис. 8.7 Кеш прямого отображения с восемью наборами

Пример 8.6 ВРЕМЕННАЯ ЛОКАЛЬНОСТЬ С КЕШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Циклы – это типичный источник временной и пространственной локальности данных в приложениях. Используя кеш с восемью ячейками, изображенный на **рис. 8.7**, покажите, чему будет равно содержимое кеша после выполнения следующего небольшого цикла на языке ассемблера RISC-V. Считайте, что изначально кеш пуст. Каким будет процент промахов?

```
      addi s0, zero, 5
      addi s1, zero, 0
LOOP: beq  s0, zero, DONE
      lw   s2, 4(s1)
      lw   s3, 12(s1)
      lw   s4, 8(s1)
      addi s0, s0, -1
      j    LOOP
DONE:
```

Решение Эта программа содержит цикл, повторяющийся пять раз. Каждая итерация содержит три обращения в память (три инструкции загрузки `lw`), всего 15 обращений. Когда цикл выполняется в первый раз, кеш пуст и данные, расположенные в оперативной памяти по адресам 0x4, 0xC и 0x8, должны быть загружены в наборы кеша 1, 3 и 2 соответственно. В следующих четырех итерациях цикла данные будут получены уже из кеша. На **рис. 8.8** показано содержимое кеша во время последнего обращения по адресу 0x4. Все теги равны нулю, потому что старшие 27 бит всех адресов равны нулю. Количество промахов кеша составляет 3 / 15 = 20 %.

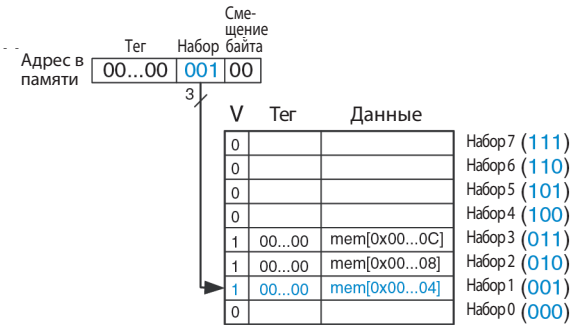


Рис. 8.8 Содержимое кеша прямого отображения

Когда два обращения к памяти по разным адресам отображаются в одну и ту же строку кеша, то возникает конфликт, и данные, загруженные во время последнего обращения, *вытесняют* (evict) из кеша данные, загруженные во время предыдущего обращения. В кеше прямого отображения в каждом наборе есть только одна строка, так что два адреса, отображаемых в одну строку, всегда вызывают конфликт. Один из таких конфликтов рассмотрен в **примере 8.7**.

Пример 8.7 КОНФЛИКТЫ ПРИ ОБРАЩЕНИИ В КЕШ-ПАМЯТЬ

Чему будет равен процент промахов при выполнении следующего цикла при наличии кеша прямого отображения емкостью 8 слов, приведенного на **рис. 8.8**? Считайте, что изначально кеш пуст.

```
      addi s0, zero, 5
      addi s1, zero, 0
LOOP: beq  s 0, zero, DONE
      lw   s2, 0x4(s1)
      lw   s4, 0x24(s1)
      addi s0, s0, -1
      j    LOOP
DONE:
```

Решение Оба адреса памяти (0x4 и 0x24) отображаются в набор 1. Во время первой итерации цикла данные по адресу 0x4 будут загружены в набор 1. Затем в тот же набор загружаются данные по адресу 0x24, вытесняя данные по адресу 0x4. Во время второй итерации все повторяется: кеш должен повторно прочитать данные по адресу 0x4, вытеснив данные по адресу 0x24. Эти два адреса конфликтуют, так что процент промахов кеша будет 100 %.

Многосекционный наборно-ассоциативный кеш

N-секционный наборно-ассоциативный кеш (*N*-way set associative cache) уменьшает количество конфликтов путем расширения набора

до N строк. Каждый адрес памяти по-прежнему отображается в строго определенный набор, но теперь он может быть отображен в любую из N строк этого набора. Можно сказать, что кеш прямого отображения – это односекционный наборно-ассоциативный кеш. Число N называют степенью ассоциативности кеша.

На **рис. 8.9** показана блок-схема аппаратной реализации наборно-ассоциативного кеша емкостью $C = 8$ слов с $N = 2$ секциями. Теперь в кеше только $S = 4$ набора вместо 8. Таким образом, для выбора нужного набора используются всего $\log_2 4 = 2$ бита, а не 3 бита, как ранее. Соответственно, размер тега увеличивается с 27 до 28 бит. Каждый набор теперь содержит две секции (2-way). Каждая секция состоит из строки (блока данных), тега и бита достоверности. Кеш читает теги и биты достоверности одновременно из обеих секций выбранного набора, после чего сравнивает их с адресом для определения попадания или промаха. Если происходит попадание в одну из секций кеша, то мультиплексор выбирает данные из этой секции и передает их процессору.

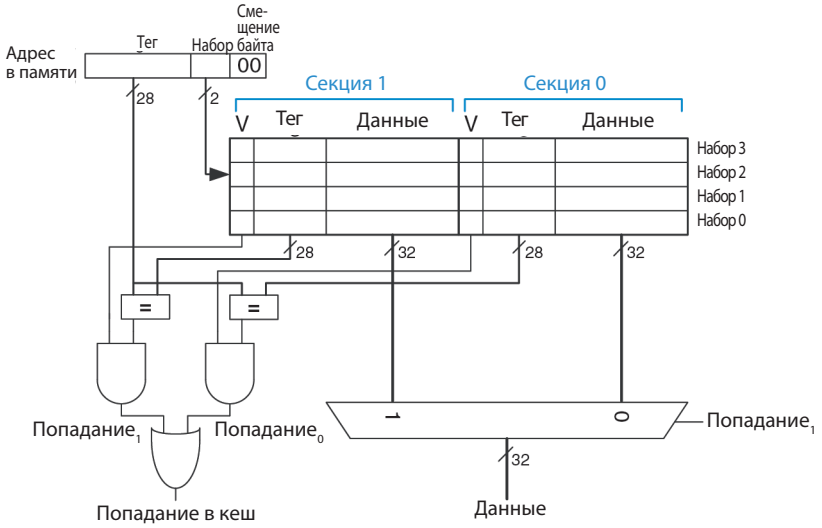


Рис. 8.9 Двухсекционный наборно-ассоциативный кеш

Наборно-ассоциативные кеши, как правило, имеют меньший процент промахов, чем кеши прямого отображения той же емкости, так как в них происходит меньше конфликтов. Но они обычно медленнее и дороже в реализации, так как необходимо использовать дополнительные компараторы и выходной мультиплексор. Кроме того, при реализации таких кешей возникает вопрос о том, какую именно секцию замещать, когда все они заняты; мы рассмотрим эту проблему в **разделе 8.3.3**. Большинство коммерческих систем сегодня используют наборно-ассоциативные кеши.

Пример 8.8 ПРОЦЕНТ ПРОМАХОВ НАБОРНО-АССОЦИАТИВНОГО КЕША

Повторите **пример 8.7**, используя двухсекционный кеш, показанный на **рис. 8.9**, емкость которого равна восьми словам.

Решение Оба обращения в память (по адресу 0x4 и по адресу 0x24) отображаются в набор 1. Но теперь кеш имеет две секции, так что он может разместить данные для этих обращений в одном наборе. Во время первой итерации цикла пустой кеш приводит к двум промахам, после чего загружает два слова данных в две секции строки 1, как показано на **рис. 8.10**. Во время следующих четырех итераций данные будут прочитаны из кеша. В результате процент промахов будет 2 / 10 = 20 %. Напомним, что для кеша прямого отображения того же размера из **примера 8.7** процент промахов был равен 100 %.

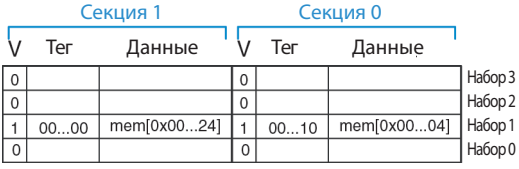


Рис. 8.10 Содержимое двухсекционного наборно-ассоциативного кеша

Полностью ассоциативный кеш

Полностью ассоциативный кеш (fully associative cache) состоит из одного набора с B секциями, где B – количество строк (блоков данных). Адрес памяти может быть отображен в строку любой из этих секций. Можно сказать, что полностью ассоциативный кеш – это B-секционный наборно-ассоциативный кеш с одним набором.

На **рис. 8.11** показан массив памяти SRAM полностью ассоциативного кеша, содержащего 8 строк. При запросе данных должны быть сделаны восемь сравнений адреса с тегами, так как данные могут быть в любой строке. Восьмивходовый мультиплексор (на рисунке не показан) выбирает соответствующую строку и подает ее на выход, если произошло попадание. Полностью ассоциативные кеши обеспечивают при прочих равных условиях минимально возможное количество конфликтов, но требуют еще больше аппаратуры для дополнительных сравнений тегов. Из-за этого они применяются лишь в относительно маленьких кешах.

Рис. 8.11 Полностью ассоциативный кеш с восемью строками

Длина строки

В предыдущих примерах мы использовали преимущества исключительно временной локальности данных, так как длина строки (т. е. размер блока данных) была равна одному слову. Чтобы воспользоваться

пространственной локальностью, в кеш-памяти используют бóльшие по размеру строки, содержащие несколько слов, расположенных последовательно в памяти.

Преимущество строк с длиной, превышающей одно слово, заключается в том, что когда случается промах кеша и требуется прочитать слово данных из памяти, то в эту строку заодно загружаются и соседние слова. Таким образом, последующие обращения с большей вероятностью приведут к попаданию в кеш из-за пространственной локальности данных. При этом увеличившаяся длина строки означает, что кеш того же размера теперь будет иметь меньшее количество самих строк. Это может привести к увеличению количества конфликтов и, соответственно, увеличить вероятность промахов кеша. Более того, потребуется больше времени на чтение данных в строку после промаха, т. к. из памяти необходимо будет прочитать не одно, а несколько слов. Время, требуемое для загрузки данных в строку кеша после промаха, называется *ценой промаха* (miss penalty). Если соседние слова данных в строке не будут использованы в дальнейшем, то усилия на их загрузку будут потрачены зря. Тем не менее большинству реальных программ увеличение длины строки приносит пользу.

На **рис. 8.12** показана блок-схема аппаратной реализации кеша прямого отображения емкостью 8 слов с длиной строки $b = 4$ слова. В кеше теперь есть только $B = C / b = 2$ строки. Так как в кеше прямого отображения количество строк и наборов совпадает, то в данном случае кеш содержит два набора, соответственно только $\log_2 2 = 1$ бит адреса используется для определения индекса (номера набора). Теперь понадобится новый мультиплексор для выбора одного из слов строки, которое и будет передано процессору. Этот мультиплексор управляется $\log_2 4 = 2$ битами адреса, которые называются *смещением в строке* (line offset или block offset). Оставшиеся 27 старших бит адреса образуют тег. На всю строку нужен всего один тег, так как слова в ней находятся по последовательным адресам.

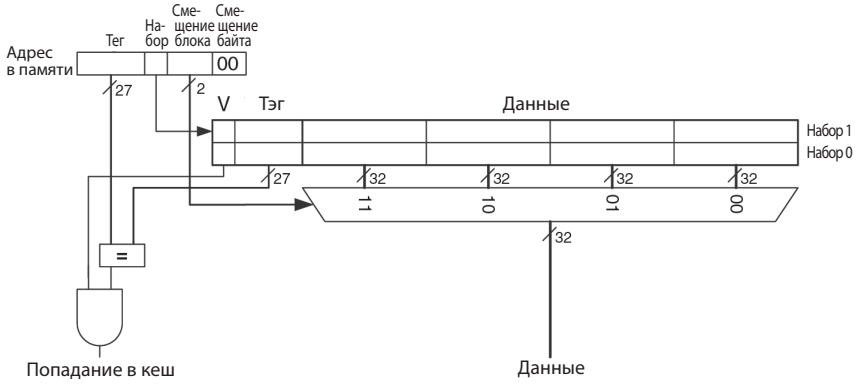


Рис. 8.12 Кеш прямого отображения с длиной строки, равной четырем словам

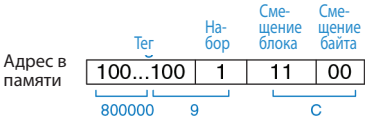


Рис. 8.13 Части адреса 0x8000009C при доступе в кеш, показанном на рис. 8.12

На **рис. 8.13** показано, на какие части делится адрес 0x8000009C при доступе в кеш прямого отображения, показанный на **рис. 8.12**. Байтовое смещение (byte offset) всегда равно нулю при доступе к словам. Следующие $\log_2 b = 2$ бита – номер слова в строке, или смещение в строке. Следующий бит выбирает один из двух наборов (set), а оставшиеся 27 бит образуют тег (tag). Следовательно, слово по адресу 0x8000009C отображается в третье слово набора 1 кеша. Принцип использования более длинных строк для использования свойства пространственной локальности данных применяется и в наборно-ассоциативных кешах.

Пример 8.9 ПРОСТРАНСТВЕННАЯ ЛОКАЛЬНОСТЬ С КЕШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Повторите **пример 8.6** для кеша прямого отображения емкостью 8 слов, длина строки которого равна четырем словам.

Решение На **рис. 8.14** показано содержимое кеша после первого обращения к памяти. Во время первой итерации цикла происходит промах кеша при обращении в память по адресу 0x4, после чего в строку кеша загружаются данные с адреса 0x0 по адрес 0xC. Все последующие обращения (как показано на рисунке для адреса 0xC) попадают в кеш. Следовательно, процент промахов будет равен $1/15 = 6{,}67\ \%$.

Рис. 8.14 Содержимое кеша с длиной строки, равной четырем словам

Промежуточные итоги

Кеш представляет собой двумерный массив. Строки этого массива называют наборами, а колонки – секциями. Каждый элемент массива содержит строку (т. е. блок данных) и связанные с ней тег и бит достоверности. Кеш характеризуется:

- емкостью C;
- длиной строки b и соответствующим количеством строк $B = C/b$;
- количеством строк в наборе (N).

В **табл. 8.2** перечислены различные способы организации кеш-памяти. Любой адрес в памяти отображается только в один набор, но соответствующие этому адресу данные могут оказаться в любой из секций этого набора.

Таблица 8.2 Способы организации кеш-памяти

Способ организации	Количество секций (*N*)	Количество наборов (*S*)
Прямого отображения	1	*B*
Наборно-ассоциативный	$1 < N < B$	B / N
Полностью ассоциативный	*B*	1

Емкость кеша, степень ассоциативности, количество наборов и длина строки обычно кратны степени двойки. Это позволяет однозначно соотносить определенные биты адреса с битами тега, индекса (номера набора) и смещения в строке.

Увеличение степени ассоциативности N обычно уменьшает процент промахов кеша, вызванных конфликтами. При этом бóльшая степень ассоциативности требует большего количества компараторов для сравнения адреса с тегами. Увеличение длины строки b позволяет использовать пространственную локальность данных для уменьшения процента промахов, но при прочих равных условиях уменьшает количество наборов и может привести к увеличению количества конфликтов. Вдобавок большая длина строки увеличивает цену промаха (miss penalty).

8.3.3. Какие данные заместить в кеш-памяти?

В кеш-памяти прямого отображения каждый адрес всегда отображается в одну и ту же строку одного и того же набора, поэтому когда нужно загрузить новые данные в набор, который уже содержит данные, то строка в наборе просто замещается на новые данные. В наборно-ассоциативной и полностью ассоциативной кеш-памяти нужно решить, какую именно из нескольких строк в наборе вытеснить. Учитывая принцип временной локальности, наилучшим вариантом было бы заменить ту строку, которая дольше всего не использовалась, потому что маловероятно, что она будет использована снова. Именно поэтому большинство кешей используют стратегию замены редко используемых данных (least recently used, LRU).

В двухсекционном наборно-ассоциативном кеше *бит использования* U (от англ. used) содержит номер той секции в наборе, которая дольше не использовалась. Каждый раз, когда происходит доступ к одной из секций набора, бит U устанавливается таким образом, чтобы указывать на другую секцию. Для наборно-ассоциативных кешей с бóльшим количеством секций отслеживать самые редко используемые строки становится сложно. Для упрощения реализации секции часто делят на две группы, а бит использования указывает на ту группу, которая дольше не использовалась. При необходимости заместить строку вытесняется случайным образом выбранная строка из той группы, которая дольше не использовалась. Такая стратегия называется *псевдо-LRU* (pseudo-LRU) и на практике достаточно хорошо работает.

Пример 8.10 СТРАТЕГИЯ ЗАМЕЩЕНИЯ LRU

Покажите содержимое двухсекционного наборно-ассоциативного кеша емкостью 8 слов после выполнения следующего кода. Используйте стратегию замещения LRU, длину строки, равную одному слову. Считайте, что изначально кеш пуст.

```
addi t0, zero, 0
lw   s1, 0x4(t0)
lw   s2, 0x24(t0)
lw   s3, 0x54(t0)
```

Решение Первые две инструкции загружают данные из памяти по адресам 0x4 и 0x24 в набор 1 кеша, как показано на **рис. 8.15 (a)**. Бит использования U = 0 показывает, что данные в секции 0 (Way 0) использовались раньше, чем в секции 1. Следующее обращение в память по адресу 0x54 также отображается в строку 1 и вытесняет дольше всего не использовавшиеся данные из секции 0, как показано на **рис. 8.15 (b)**. Бит использования при этом устанавливается в 1, указывая, что теперь именно данные в секции 1 не использовались дольше.

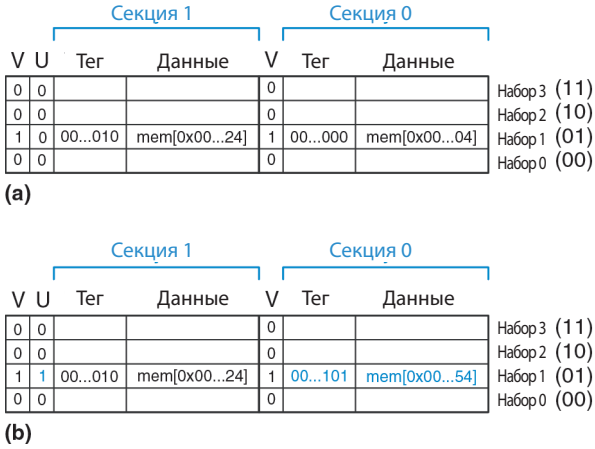


Рис. 8.15 Двухсекционный кеш со стратегией замещения LRU

8.3.4. Улучшенная кеш-память

В современных системах для сокращения времени доступа к памяти используется несколько уровней кеша. В этом разделе мы рассмотрим производительность двухуровневой системы кеширования и выясним, как длина строки, ассоциативность и емкость кеша влияют на частоту промахов. Также мы рассмотрим, как кеш-память ведет себя при записи данных в память с использованием стратегий сквозной (write-though) или отложенной (write-back) записи.

Многоуровневые кеши

Чем больше размер кеша, тем больше вероятность, что интересующие нас данные в нем будут найдены, и, следовательно, тем меньше

у него будет частота промахов. Но большой кеш обычно медленнее, чем маленький, поэтому в современных системах используются как минимум два уровня кеша, как показано на **рис. 8.16**. Кеш первого уровня (L1) достаточно мал, чтобы обеспечить время доступа в один или два такта. Кеш второго уровня (L2) тоже сделан на основе SRAM, но больше по размеру и поэтому медленнее, чем кеш L1. Сначала процессор ищет данные в кеше L1, а если происходит промах – то в кеше L2. Если и там происходит промах, то процессор обращается за данными к оперативной памяти. Многие современные системы используют еще больше уровней кеша в иерархии памяти, так как доступ к оперативной памяти чрезвычайно медленный.

Пример 8.11 СИСТЕМА С КЕШЕМ L2

Используйте систему, показанную на **рис. 8.16**, со временем доступа 1, 10 и 100 циклов для кеша L1, кеша L2 и основной памяти соответственно. Предположим, что кеши L1 и L2 имеют процент промахов 5 % и 20 % соответственно. В частности, из 5 % обращений без кеша L1 20 % из них также не попадают в кеш L2. Какое среднее время доступа к памяти (average memory access time, AMAT)?

Решение При каждом обращении к памяти процессор сначала ищет запрошенные данные в кеше L1. Когда происходит промах (5 % случаев), процессор ищет их в кеше L2. Если снова возникает промах кеша (20 % случаев), то процессор обращается за данными в оперативную память. Используя формулу (8.2), мы можем вычислить среднее время доступа к памяти как

Рис. 8.16 Иерархия памяти с двумя уровнями кеша

$$1 \text{ такт} + 0{,}05\,(10 \text{ тактов} + 0{,}2\,(100 \text{ тактов})) = 2{,}5 \text{ такта}.$$

Доля промахов в кеше L2 выше, поскольку до него доходят лишь «трудные» обращения в память – те, которые уже привели к промаху кеша L1. Если бы все обращения шли непосредственно в кеш L2, доля его промахов была бы около 1 %.

Сокращение частоты промахов

Доля промахов кеша можно сократить, изменяя его емкость, длину строки и/или ассоциативность. Для этого сначала необходимо разобраться с причинами промахов. Промахи кеша делятся на *неизбежные промахи* (compulsory misses), *промахи из-за недостаточной емкости* (capacity misses) и *промахи из-за конфликтов* (conflict misses). Первое обращение к строке кеша всегда приводит к неизбежному промаху, так как эту строку нужно прочесть из оперативной памяти хотя бы один раз независимо от архитектуры кеша. Промахи из-за недостаточной емкости происходят, когда кеш слишком мал для хранения всех одновременно используемых данных. Промахи из-за конфликтов случаются, если не-

сколько адресов памяти отображаются на один и тот же набор кеша и выталкивают из него данные, которые все еще нужны.

Изменение параметров кеша может повлиять на частоту одного или нескольких типов промахов. Например, увеличение размера кеша может сократить промахи из-за конфликтов и промахи из-за недостатка емкости, но никак не повлияет на количество неизбежных промахов. С другой стороны, увеличение длины строки может сократить количество неизбежных промахов (благодаря локальности данных), но одновременно может увеличить частоту промахов из-за конфликтов, поскольку большее количество адресов будет отображаться на один и тот же набор, увеличивая вероятность конфликтов.

Системы памяти настолько сложны, что лучший способ оценивать их производительность – это запускать тестовые программы, варьируя параметры кеша. На **рис. 8.17** изображен график зависимости частоты промахов от размера кеша и степени ассоциативности для набора тестовых программ SPEC2000. Небольшое количество неизбежных промахов показано темным цветом вдоль оси X и не зависит от емкости кеша. С другой стороны, как и ожидалось, с увеличением емкости кеша частота промахов из-за недостатка емкости сокращается. Увеличение ассоциативности, особенно для кешей небольшого размера, сокращает количество промахов из-за конфликтов, показанных вдоль верхней части кривой. При этом ассоциативность свыше четырех или восьми секций приводит лишь к незначительному сокращению частоты промахов.

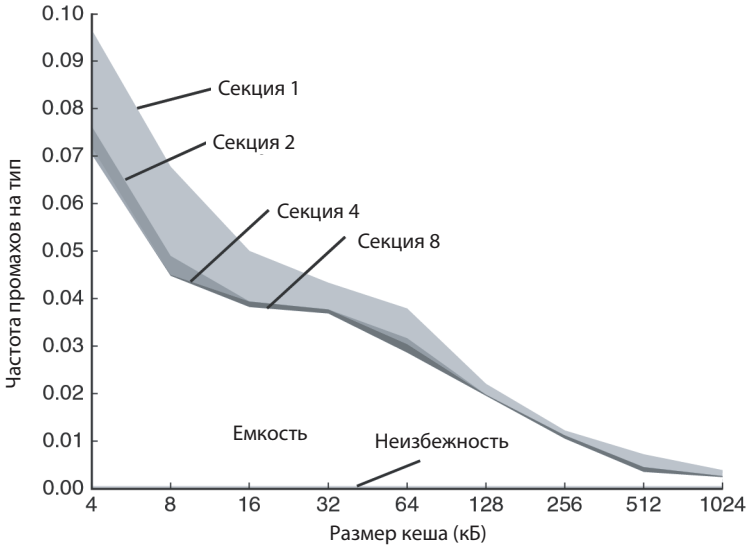


Рис. 8.17 Зависимость частоты промахов от размера и ассоциативности кеша на тестах SPEC2000
(график из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012, с разрешения авторов)

Как уже говорилось, частоту промахов можно уменьшить, используя пространственную локальность данных с помощью более длинных строк кеша. Но при прочих равных условиях с увеличением длины строки в кеше уменьшается количество наборов, что увеличивает вероятность конфликтов. На **рис. 8.18** представлена зависимость частоты промахов от длины строки в байтах (block size) для кешей разной емкости. Для небольших кешей, таких как кеш емкостью 4 Кбайта, длина строки свыше 64 байт увеличивает частоту промахов из-за конфликтов. Для кешей большей емкости длина строки свыше 64 байт не влияет на частоту промахов. При этом большая длина строки все же может вызвать увеличение времени выполнения из-за более высокой цены промаха (miss penalty) – времени, требуемого для выборки отсутствующей строки кеша из оперативной памяти.

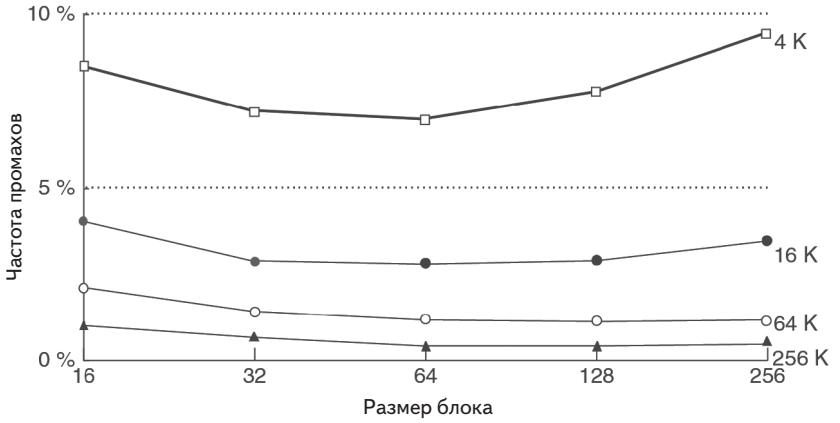


Рис. 8.18 Зависимость частоты промахов от длины строки и размера кеша на тестах SPEC92
(график из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012, с разрешения авторов)

Стратегии записи данных в память

В предыдущих разделах мы рассматривали чтение из памяти. Запись в память выполняется примерно так же, как и чтение. При выполнении команды сохранения данных процессор сначала проверяет кеш. В случае промаха кеша соответствующая строка выбирается из оперативной памяти в кеш, а затем в нее записывается нужное слово. В случае попадания в кеш слово просто записывается в строку.

Кеши делятся на два типа – со *сквозной записью* (write-through) и с *отложенной записью* (write-back). В кеше со сквозной записью данные, записываемые в кеш, одновременно записываются и в оперативную память. В кеше с отложенной записью у каждой строки есть бит изменения *D* (от англ. *dirty*). Если в строку производилась запись, то этот бит равен 1, в противном случае он равен 0. Измененные строки

записываются обратно в оперативную память только тогда, когда они вытесняются из кеша. В кеше со сквозной записью биты изменения не нужны, но такой кеш обычно приводит к большему количеству операций записи в память, чем кеш с отложенной записью. Из-за того, что время обращения к оперативной памяти очень велико, в современных системах обычно используют кеши с отложенной записью.

Пример 8.12 СКВОЗНАЯ И ОТЛОЖЕННАЯ ЗАПИСЬ

Допустим, что длина строки кеша – четыре слова. Сколько обращений к оперативной памяти потребуется при выполнении кода, приведенного ниже, если используется стратегия сквозной записи, и сколько, если используется отложенная запись?

```
addi t5, zero, 0
sw   t1, 0(t5)
sw   t2, 12(t5)
sw   t3, 8(t5)
sw   t4, 4(t5)
```

Решение Все четыре команды сохранения изменяют одну и ту же строку кеша. При сквозной записи каждая команда сохраняет слово в оперативную память, соответственно, потребуется четыре обращения к памяти. При отложенной записи потребуется только одно обращение – тогда, когда эта строка будет вытеснена из кеша.

8.4. Виртуальная память

Большинство современных вычислительных систем в качестве нижнего уровня в иерархии памяти используют жесткие диски, представляющие собой магнитные или твердотельные запоминающие устройства (**рис. 8.4**). По сравнению с идеальной памятью, которая должна быть быстрой, дешевой и большой, жесткий диск имеет большой объем и недорого стоит, но медленно работает. Жесткий диск обеспечивает намного больший объем, чем недорогая оперативная память (DRAM). Но если существенная часть обращений к памяти осуществляется к жесткому диску, скорость работы всей системы сильно снижается. Вы могли сталкиваться с этой проблемой на персональном компьютере, если одновременно запускали слишком много программ.

На **рис. 8.19** показан магнитный *жесткий диск* со снятой крышкой. Как следует из названия, жесткий диск состоит из одной или нескольких *пластин*, каждой из которых касается *головка считывания-записи*, расположенная на конце длинного треугольного кронштейна. Головка перемещается в правильное положение на диске и считывает или записывает информацию при помощи магнитного поля в то время, пока диск вращается под ней. Головка ищет правильное положение на диске в те-

чение нескольких миллисекунд – это быстро с точки зрения человека, но в миллионы раз медленнее, чем скорость работы процессора. Жесткие диски все чаще заменяются твердотельными накопителями (SSD), которые выполняют чтение на порядок быстрее (**рис. 8.4**) и не склонны к механическим поломкам.

Рис. 8.19 Жесткий диск

Цель включения жесткого диска в иерархию памяти – с минимальными затратами создать видимость памяти большого объема, одновременно обеспечивая для большинства обращений к памяти скорость доступа, равную скорости более быстрых типов памяти. Например, компьютер с оперативной памятью на 16 Гбайт может обеспечить видимость наличия 128 Гбайт оперативной памяти, используя для этого жесткий диск. В этом случае большая память объемом 128 Гбайт называется *виртуальной памятью*, а меньшая память объемом 16 Гбайт называется *физической памятью*. В этом разделе мы будем использовать термин *физическая память* для обозначения оперативной памяти компьютера.

Программы могут обращаться к данным в любом месте виртуальной памяти, поэтому они должны использовать виртуальные адреса, которые определяют расположение данных в виртуальной памяти. Физическая память хранит последние запрошенные из виртуальной памяти блоки данных. Таким образом, физическая память выступает в роли кеша для виртуальной памяти, то есть большинство обращений происходят к быстрой физической памяти (DRAM), и в то же время программа имеет доступ к большей по объему виртуальной памяти.

Подсистемы виртуальной памяти используют другие термины для тех же самых принципов кеширования, которые были рассмотрены в **раз-**

Компьютер с 32-битной адресацией имеет доступ к 2^{32} байтам = 4 Гбайтам памяти. Это одна из причин перехода к 64-битным компьютерам, которые могут получать доступ к памяти большего объема.

деле 8.3. В табл. 8.3 приведены сходные термины. Виртуальная память разделена на *виртуальные страницы*, обычно размером 4 Кбайт. Физическая память аналогичным образом разделена на *физические страницы*. Размер виртуальных и физических страниц одинаков. Виртуальная страница может располагаться в физической памяти (DRAM) или на жестком диске. Например, на рис. 8.20 показана виртуальная память, которая больше физической памяти. Прямоугольники обозначают страницы. Некоторые виртуальные страницы расположены в физической памяти, а некоторые – на жестком диске. Процесс преобразования виртуального адреса в физический называется *трансляцией адреса*. Если процессор обращается к виртуальному адресу, которого нет в физической памяти, происходит страничная ошибка (page fault), и операционная система загружает соответствующую страницу с жесткого диска в физическую память.

Таблица 8.3 Соответствие терминов кеша и виртуальной памяти

Кеш	Виртуальная память
Строка	Страница
Длина строки	Размер страницы
Смещение относительно начала строки	Смещение относительно начала страницы
Промах	Страничная ошибка
Тег	Номер виртуальной страницы

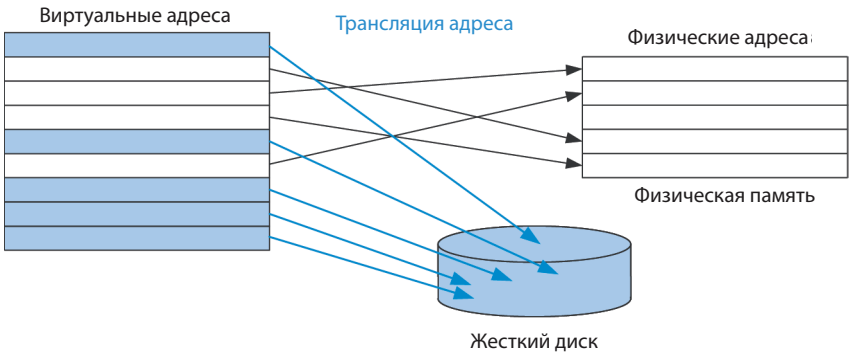


Рис. 8.20 Виртуальные и физические страницы

Чтобы избежать страничных ошибок, вызванных конфликтами, любая виртуальная страница может отображаться на любую физическую страницу. Другими словами, физическая память работает как полностью ассоциативный кеш для виртуальной памяти. В традиционном полностью ассоциативном кеше каждая секция содержит компаратор, который проверяет старшие биты адреса на соответствие тегу, чтобы определить,

находятся ли там нужные данные. В аналогичной системе виртуальной памяти каждой физической странице нужен был бы компаратор, чтобы сверять старшие биты виртуальных адресов с тегом и определять, отображается ли виртуальная страница на эту физическую страницу.

На практике виртуальная память имеет настолько много физических страниц, что обеспечить компаратором каждую страницу было бы чересчур дорого. Вместо этого в подсистемах виртуальной памяти используется трансляция адреса при помощи *таблицы страниц* (page table). Таблица страниц содержит запись для каждой виртуальной страницы, указывающую ее расположение в физической памяти или на жестком диске. Каждая команда загрузки или сохранения требует доступа к таблице страниц с последующим доступом к физической памяти. Обращение к таблице страниц позволяет транслировать виртуальный адрес, используемый программой, в физический адрес. Затем физический адрес используется для фактического чтения или записи данных.

Таблица страниц обычно настолько велика, что сама находится в физической памяти. Таким образом, каждая команда загрузки или сохранения данных включает два обращения к физической памяти: обращение к таблице страниц и собственно обращение к данным. Чтобы ускорить трансляцию адреса, используется *буфер ассоциативной трансляции* (translation lookaside buffer, TLB), который содержит наиболее часто используемые записи таблицы страниц.

В оставшейся части этого раздела мы более подробно рассмотрим трансляцию адресов, таблицы страниц и TLB.

8.4.1. Трансляция адресов

В системах с виртуальной памятью программы используют виртуальные адреса и поэтому имеют доступ к памяти большого объема. Компьютер должен транслировать эти виртуальные адреса, чтобы либо найти соответствующий адрес в физической памяти, либо получить страничную ошибку и загрузить данные с жесткого диска.

Вспомните, что виртуальная память и физическая память разделены на страницы. Старшие биты виртуального и физического адресов определяют номер виртуальной и физической страниц соответственно. Младшие биты определяют положение слова внутри страницы и называются смещением относительно начала страницы.

На **рис. 8.21** показана страничная организация подсистем виртуальной памяти объемом 2 Гбайта и физической памяти объемом 128 Мбайт, разделенных на страницы по 4 Кбайт. RISC-V использует 32-битную адресацию. Так как виртуальная память имеет объем 2 Гбайт = 2^{31} байт, то используются только младшие 31 бит виртуального адреса, а старший бит всегда равен нулю. Аналогично, так как физическая память имеет объем 128 Мбайт = 2^{27} байт, используются только младшие 27 бит физического адреса, а старшие 5 бит всегда равны нулю.

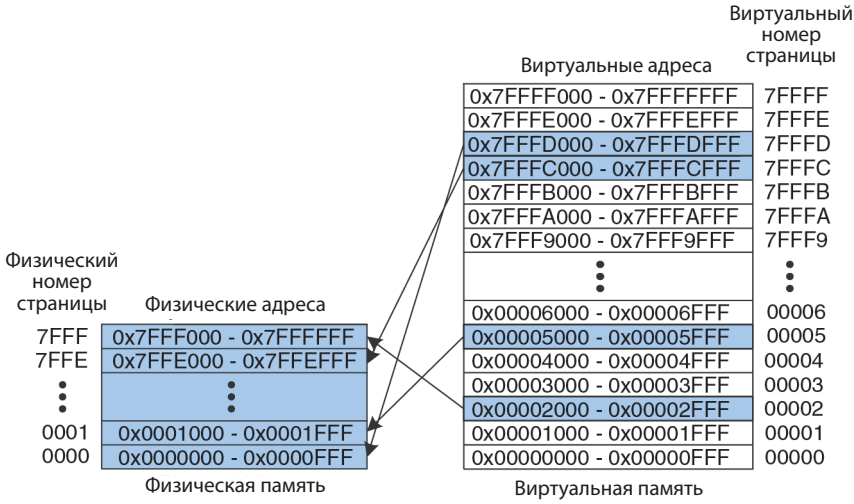


Рис. 8.21 Физические и виртуальные страницы

Поскольку размер страницы составляет 4 Кбайт = 2^{12} байт, существует $2^{31}/2^{12} = 2^{19}$ виртуальных страниц и $2^{27}/2^{12} = 2^{15}$ физических страниц. Таким образом, номера страниц виртуальной и физической памяти состоят из 19 и 15 бит соответственно. В любой момент времени физическая память может хранить максимум 1/16 от количества страниц виртуальной памяти. Остальные виртуальные страницы хранятся на жестком диске.

На **рис. 8.21** показано, как виртуальная страница 5 отображается на физическую страницу 1, виртуальная страница 0x7FFFC отображается на физическую страницу 0x7FFE и т. д. Например, виртуальный адрес 0x53F8 (смещение 0x3F8 от начала виртуальной страницы номер 5) отображается на физический адрес 0x13F8 (смещение 0x3F8 от начала физической страницы номер 1). Младшие 12 бит виртуального и физического адресов одинаковы (0x3F8) и определяют смещение от начала виртуальной и физической страниц. Таким образом, чтобы получить физический адрес из виртуального, необходимо транслировать только номер страницы.

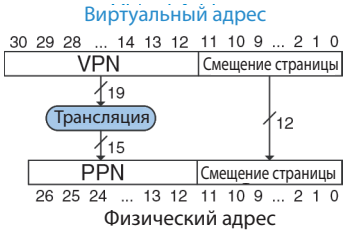


Рис. 8.22 Трансляция виртуального адреса в физический

На **рис. 8.22** показан процесс трансляции виртуального адреса в физический. Младшие 12 бит определяют смещение от начала страницы и не нуждаются в трансляции. Старшие 19 бит виртуального адреса определяют номер виртуальной страницы (virtual page number, VPN) и транслируются в 15-битный номер физической страницы (physical page number, PPN). В следующих двух разделах рассказывается, как для трансляции адресов используются таблицы страниц и TLB.

Пример 8.13 ТРАНСЛЯЦИЯ ВИРТУАЛЬНОГО АДРЕСА В ФИЗИЧЕСКИЙ

Найдите физический адрес, соответствующий виртуальному адресу 0x247C, используя подсистему виртуальной памяти, показанную на **рис. 8.21**.

Решение 12 бит, обозначающих смещение от начала страницы (0x47C), не нуждаются в трансляции. Оставшиеся 19 бит виртуального адреса определяют номер виртуальной страницы. Это означает, что виртуальный адрес 0x247C находится внутри виртуальной страницы 0x2. Согласно **рис. 8.21**, виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C.

8.4.2. Таблица страниц

Процессор использует таблицу страниц для трансляции виртуальных адресов в физические. Таблица страниц содержит отдельную запись для каждой виртуальной страницы. Эта запись содержит номер физической страницы и бит достоверности (valid bit). Если бит достоверности равен 1, виртуальная страница отображается на физическую страницу, номер которой указан в записи. В обратном случае виртуальная страница находится на жестком диске.

Поскольку таблица страниц велика, она хранится в физической памяти. Предположим, что она хранится в виде непрерывного массива, как показано на **рис. 8.23**. Эта таблица страниц содержит информацию об отображении страниц, показанных на **рис. 8.21**. Номера строк в таблице страниц соответствуют номерам виртуальных страниц (VPN). Например, строка номер 5 определяет, что виртуальная страница 5 отображается на физическую страницу 1. Запись 6 недействительна (бит достоверности V = 0), то есть виртуальная страница 6 расположена на жестком диске, а не в физической памяти.

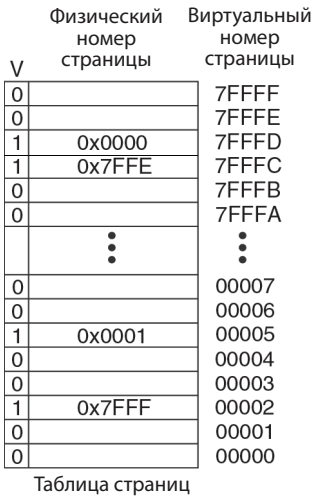


Рис. 8.23 Таблица страниц для случая, показанного на рис. 8.21

Пример 8.14 ИСПОЛЬЗОВАНИЕ ТАБЛИЦЫ СТРАНИЦ ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Найдите физический адрес, соответствующий виртуальному адресу 0x247C, используя таблицу страниц, приведенную на **рис. 8.23**.

Решение На **рис. 8.24** показана трансляция виртуального адреса в физический для виртуального адреса 0x247C. 12 бит, обозначающих смещение от начала страницы, не нуждаются в трансляции. Оставшиеся 19 бит виртуального адреса – это номер виртуальной страницы, 0x2, они определяют номер в таблице страниц. Таблица страниц содержит информацию о том, что виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C. Младшие 12 бит одинаковы для физического и виртуального адресов.

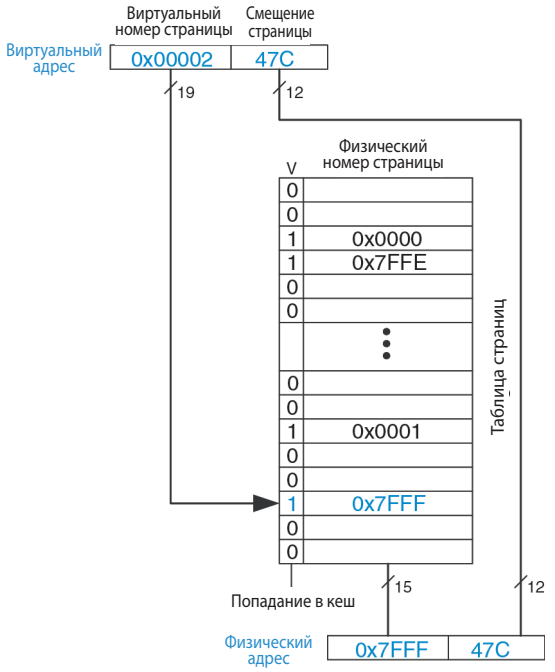


Рис. 8.24. Трансляция адреса при помощи таблицы страниц

Таблица страниц может храниться в любом месте физической памяти, ее расположение определяется операционной системой. Процессор обычно использует выделенный регистр, называемый *регистром таблицы страниц*, для хранения ее базового адреса.

Чтобы выполнить операцию загрузки или сохранения данных, процессор должен сначала транслировать виртуальный адрес в физический, а затем обратиться к физической памяти, используя полученный физический адрес. Процессор извлекает номер виртуальной страницы из виртуального адреса и прибавляет его к содержимому регистра таблицы страниц, чтобы найти физический адрес соответствующей записи в таблице страниц, расположенной в физической памяти. Затем процессор считывает эту запись и получает номер физической страницы. Если запись действительна, т. е. бит достоверности равен 1, то процессор объединяет номер физической страницы и смещение и получает физический адрес. Наконец, он читает или записывает данные, используя этот физический адрес. Поскольку таблица страниц тоже хранится в физической памяти, каждая команда загрузки или сохранения требует два обращения к физической памяти.

TLB организован как полностью ассоциативный кеш и обычно хранит от 16 до 512 записей. Каждая запись в TLB хранит номер виртуальной страницы и соответствующий ей номер физической страницы.

Обращение к TLB происходит по номеру виртуальной страницы. Если происходит попадание в TLB, то возвращается соответствующий номер физической страницы. В противном случае процессор должен прочитать нужную запись из таблицы страниц в физической памяти. Буфер ассоциативной трансляции разрабатывают таким образом, чтобы он был небольшого размера и чтобы доступ к нему занимал менее одного такта. Даже при этом доля попаданий в него обычно превышает 99 %. TLB уменьшает количество обращений к памяти, требуемое для большинства команд загрузки и сохранения, с двух до одного.

Пример 8.15 ИСПОЛЬЗОВАНИЕ TLB ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Рассмотрим подсистему виртуальной памяти, приведенную на **рис. 8.21**. Используйте TLB с двумя записями или объясните, почему необходимо обращение к таблице страниц, чтобы осуществить трансляцию виртуальных адресов 0x247C и 0x5FB0 в физические адреса. Предположим, что TLB хранит корректные записи для виртуальных страниц 0x2 и 0x7FFFD.

Решение На **рис. 8.25** показан TLB с двумя записями и входящим запросом на трансляцию виртуального адреса 0x247C. Из поступившего на вход TLB адреса извлекается номер виртуальной страницы, равный 0x2, после чего этот номер сравнивается с номерами виртуальных страниц, находящимися в записях таблицы. Совпадение номеров обнаружено для записи 0 (Entry 0), при этом запись действительна ($V = 1$), поэтому произошло попадание в TLB. Транслированный физический адрес представляет собой номер физической страницы из записи с совпавшим номером, равный 0x7FFF, объединенный со смещением, скопированным из виртуального адреса. Как всегда, смещение не требует трансляции.

Запрос на трансляцию виртуального адреса 0x5FB0 приводит к промаху TLB, поэтому он пересылается для трансляции с помощью таблицы страниц.

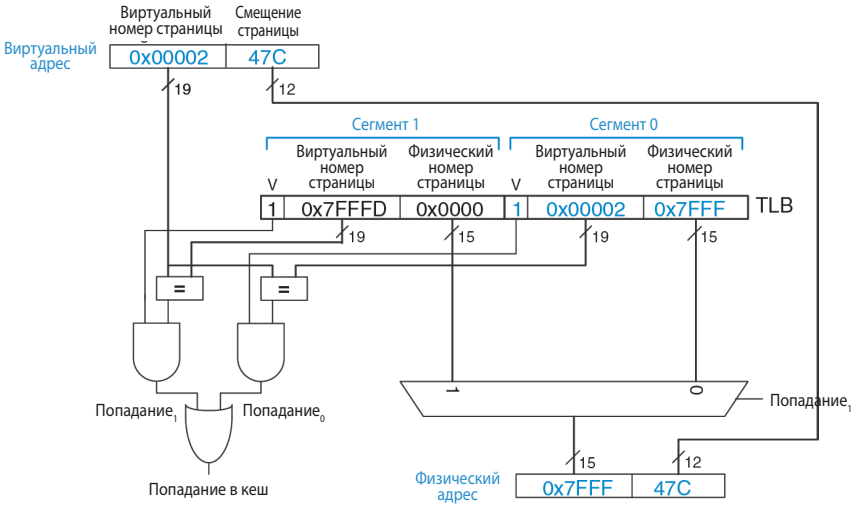


Рис. 8.25. Трансляция адреса с использованием TLB

8.4.4. Защита памяти

До сих пор мы рассматривали применение виртуальной памяти для создания видимости наличия быстрой, недорогой и большой по объему памяти. Не менее важной причиной использования виртуальной памяти является обеспечение защиты нескольких одновременно выполняемых программ друг от друга.

Как вы, возможно, знаете, современные компьютеры обычно выполняют несколько программ или процессов одновременно. Все эти программы одновременно присутствуют в физической памяти. В хорошо спроектированной компьютерной системе программы должны быть защищены друг от друга, чтобы работа одной программы не могла нарушить работу другой программы. Точнее, ни одна программа не должна иметь доступ к памяти другой программы без разрешения. Это называется *защитой памяти*.

Системы виртуальной памяти обеспечивают защиту памяти, предоставляя каждой программе персональное виртуальное адресное пространство. Каждая программа может использовать столько памяти в пределах виртуального пространства, сколько необходимо, но только часть виртуального адресного пространства находится в физической памяти в каждый момент времени. Каждая программа может полностью использовать свое виртуальное адресное пространство, не беспокоясь о том, где расположены остальные программы. При этом программа имеет доступ только к тем физическим страницам, информация о которых находится в ее таблице страниц. Таким образом, программа не может случайно или преднамеренно получить доступ к физическим страницам другой программы, поскольку они не отображены в ее таблице страниц. Иногда несколько программ должны иметь доступ к общим командам или данным. Для этого операционная система добавляет к каждой записи в таблице страниц несколько служебных битов, позволяющих определить, какие именно программы могут изменять общие (разделяемые, англ.: shared) физические страницы.

8.4.5. Стратегии замещения страниц

Подсистемы виртуальной памяти используют стратегию *отложенной записи* (write-back) и стратегию вытеснения редко используемых страниц (least recently used, LRU) для замещения страниц в физической памяти. Стратегия *сквозной записи* (write-though), при которой каждая запись в физическую память приводит заодно и к записи на жесткий диск, была бы непрактична, потому что команды сохранения выполнялись бы со скоростью жесткого диска, а не со скоростью процессора (миллисекунды вместо наносекунд). При стратегии обратной записи физическая страница записывается обратно на жесткий диск только тогда, когда она

вытесняется из физической памяти. Процесс записи физической страницы обратно на жесткий диск и размещение на ее месте другой виртуальной страницы называется *замещением страниц* (paging), а жесткий диск в подсистемах виртуальной памяти иногда называется *пространством подкачки* (swap). Когда происходит страничная ошибка, процессор замещает одну из редко используемых физических страниц на отсутствующую виртуальную страницу, вызвавшую страничную ошибку. Такая реализация требует наличия в каждой записи таблицы страниц двух дополнительных служебных битов: бита изменения *D* (dirty bit) и бита доступа *U* (use bit).

Бит изменения равен 1, если любая из команд сохранения изменила содержимое физической страницы с момента ее считывания с жесткого диска. Когда физическая страница с битом изменения, равным 1, замещается на новую, то ее необходимо записать обратно на жесткий диск. Если же бит изменения замещаемой страницы равен 0, то точная копия этой страницы и так уже находится на жестком диске, поэтому записывать ее туда не имеет смысла.

Бит доступа равен 1, если к физической странице недавно обращались. Как и в случае кеш-памяти, точный алгоритм LRU было бы слишком сложно реализовать. Вместо этого операционная система использует приближенный алгоритм вытеснения редко используемых страниц, периодически сбрасывая биты доступа в таблице страниц. Если к странице обращаются, то бит доступа устанавливается в 1. При возникновении страничной ошибки операционная система находит страницу с U = 0 и вытесняет ее из физической памяти. Таким образом, замещается не обязательно самая редко используемая страница, но одна из нескольких относительно редко используемых страниц.

8.4.6. Многоуровневые таблицы страниц

Таблицы страниц могут занимать большой объем физической памяти. Например, для показанной ранее таблицы страниц при размере виртуальной памяти, равном 2 Гбайт, и размере страниц, равном 4 Кбайт, потребуется 2^{19} записей. Если размер каждой записи равен 4 байтам, то размер таблицы страниц равен $2^{19} \times 2^2$ байт = 2^{21} байт = 2 Мбайта.

Чтобы сэкономить физическую память, таблицы страниц можно поделить на несколько уровней (обычно их два). Таблица страниц первого уровня всегда хранится в физической памяти. Она указывает, в каком месте виртуальной памяти хранятся маленькие таблицы страниц второго уровня. Каждая таблица страниц второго уровня хранит информацию о некотором диапазоне виртуальных страниц. Если какой-то диапазон виртуальных адресов не используется, то соответствующая таблица страниц второго уровня может быть сохранена на жесткий диск, чтобы не занимать место в физической памяти.

В двухуровневой таблице страниц номер виртуальной страницы разделен на две части: *номер таблицы страниц* (page table number) и *смещение в таблице страниц* (page table offset), как показано на **рис. 8.26**. Номер таблицы страниц указывает на номер строки в таблице первого уровня, которая должна всегда находиться в физической памяти. Запись в таблице страниц первого уровня содержит базовый адрес таблицы страниц второго уровня или, если $V = 0$, указывает, что ее необходимо загрузить с жесткого диска. Смещение в таблице страниц указывает на номер строки в таблице второго уровня. Оставшиеся 12 бит виртуального адреса – это, как и ранее, смещение от начала страницы размером $2^{12} = 4$ Кбайт.

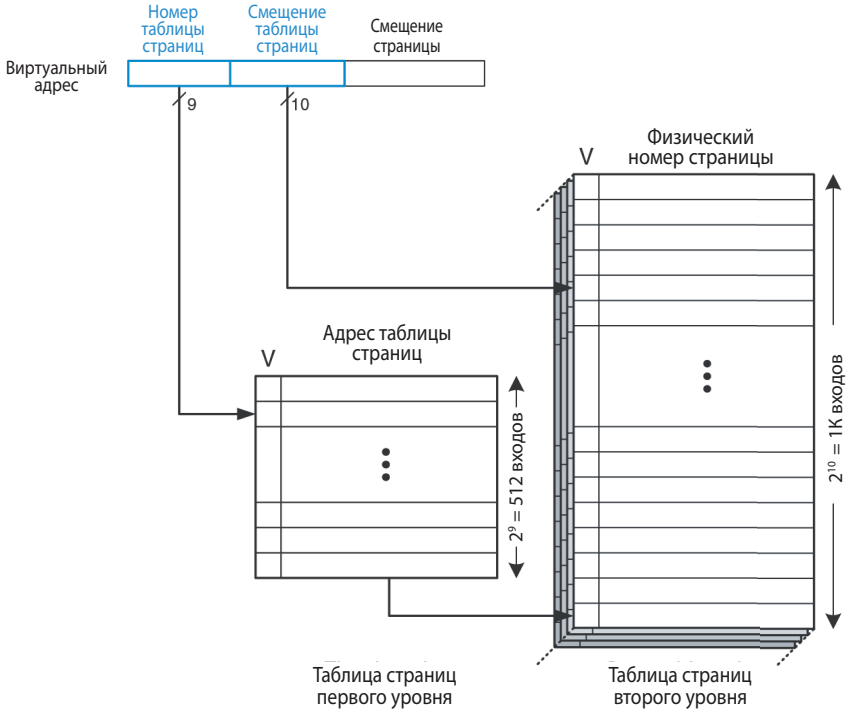


Рис. 8.26 Иерархические таблицы страниц

На **рис. 8.26** 19-битный номер виртуальной страницы разделен на две части по 9 и 10 бит, определяющие номер таблицы страниц и смещение в таблице страниц соответственно. Таким образом, таблица страниц первого уровня содержит $2^9 = 512$ записей. Каждая из 512 таблиц страниц второго уровня содержит $2^{10} = 1024$ записи.

Если размер каждой записи в таблицах страниц первого и второго уровней равен 32 битам (4 байтам) и только две таблицы страниц второго уровня присутствуют в физической памяти одновременно, то иерархиче-

ская таблица страниц занимает всего $(512 \times 4$ байт$) + 2 \times (1024 \times 4$ байт$) = 10$ Кбайт физической памяти. Очевидно, что двухуровневая таблица страниц занимает лишь малую часть физической памяти, необходимой для хранения всей одноуровневой таблицы страниц (2 Мбайта). Недостаток двухуровневой таблицы состоит в том, что при промахе TLB будет необходимо выполнить на одно обращение к памяти больше.

Пример 8.16 ИСПОЛЬЗОВАНИЕ МНОГОУРОВНЕВОЙ ТАБЛИЦЫ СТРАНИЦ

На **рис. 8.27** показан возможный вариант содержимого двухуровневой таблицы страниц, изображенной на **рис. 8.26**. Показано содержимое только одной таблицы страниц второго уровня. Используя эту двухуровневую таблицу, опишите, что происходит при обращении к виртуальному адресу 0x003FEFB0.

Решение Как и всегда, требуется транслировать только номер виртуальной страницы. Старшие девять бит виртуального адреса – это номер таблицы страниц, равный 0x0. Он определяет номер строки в таблице первого уровня. Соответствующая запись в таблице первого уровня указывает, что нужная таблица страниц второго уровня уже располагается в памяти ($V = 1$), а ее физический адрес равен 0x2375000.

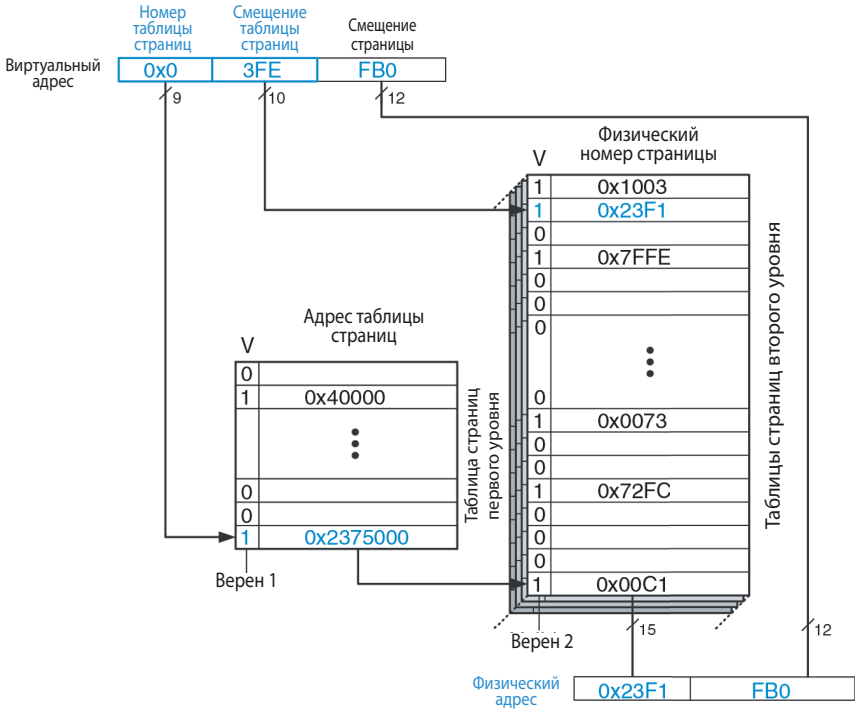


Рис. 8.27 Трансляция адреса с использованием двухуровневой таблицы страниц

Следующие 10 бит виртуального адреса, равные 0x3FE, – это смещение в таблице страниц, которое равно номеру строки в таблице второго уровня. В таблице второго уровня 1024 записи, а строки пронумерованы снизу вверх. Таким образом, запись 0x3FE в таблице страниц второго уровня – вторая сверху. Она указывает, что виртуальная страница тоже находится в физической памяти ($V = 1$), а номер физической страницы равен 0x23F1. Физический адрес получается путем объединения номера физической страницы и смещения от начала страницы и равен 0x23F1FB0.

8.5. Заключение

Организация системы памяти – важный фактор, влияющий на производительность вычислительной системы. Различные технологии производства памяти, такие как SRAM, DRAM и жесткие диски, позволяют найти компромисс между емкостью, скоростью работы и ценой памяти. В этой главе мы рассмотрели организацию иерархии памяти, включающую кеш-память и виртуальную память, которая позволяет разработчикам приблизиться к идеалу – большой, быстрой и дешевой памяти. Оперативная память обычно использует динамическую память (DRAM) и работает существенно медленнее, чем процессор. Кеш, который хранит часто используемые данные в гораздо более быстрой статической памяти SRAM, используется для уменьшения времени доступа к оперативной памяти. Виртуальная память позволяет увеличить доступный объем памяти, используя жесткий диск, на котором располагаются данные, не помещающиеся в оперативную память. Кеш и виртуальная память требуют дополнительной аппаратуры и усложняют компьютерную систему, но чаще всего их преимущества перевешивают недостатки. Во всех современных персональных компьютерах используются кеш и виртуальная память. Большинство процессоров также используют интерфейс памяти для обращения к устройствам ввода-вывода. Такой подход называется отображаемым в память вводом-выводом (memory-mapped I/O, MMIO). При подобном подходе для работы с внешними устройствами программы пользуются командами загрузки и сохранения данных. Об этом пойдет речь в **главе 9**.

Эпилог

Эта глава подводит нас к завершению путешествия по миру цифровых систем. Мы надеемся, что в данной книге мы смогли показать не только красоту и увлекательность искусства их проектирования, но и дать инженерные знания. Вы изучили, как разрабатывать комбинационную и последовательную логику на уровне схем и языков описания аппаратуры. Вы познакомились с более крупными строительными блоками, такими как мультиплексоры, АЛУ и память. Компьютеры – одно из наиболее занимательных приложений цифровых систем. Вы изучили, как програм-

мировать процессор RISC-V на его родном языке ассемблера и как построить процессор и систему памяти из цифровых строительных блоков. В процессе чтения вы наблюдали применение принципов абстракции, дисциплины, иерархии, модульности и регулярности. С их помощью мы сложили пазл внутреннего устройства микропроцессора. От мобильных телефонов до цифрового телевидения, от марсоходов до медицинских систем визуализации – наш мир становится все более и более цифровым.

Представьте, какую цену был бы готов, как Фауст, заплатить Чарльз Бэббидж, чтобы узнать все это полтора столетия назад. Он мечтал всего лишь вычислять математические таблицы с механической точностью. Сегодняшние цифровые системы вчера были фантастикой. Мог бы Дик Трейси[1] слушать iTunes на своем телефоне? Запустил бы Жюль Верн в космос навигационные спутники? Мог бы Гиппократ лечить с помощью МРТ? В то же время оруэлловский кошмар повсеместной государственной слежки становится все более реальным день ото дня. Хакеры и правительства ведут необъявленные кибервойны, атакуя промышленную инфраструктуру и финансовые системы. Страны-изгои разрабатывают ядерное оружие с помощью ноутбуков, более мощных, чем суперкомпьютеры, занимавшие целые машинные залы и использовавшиеся при расчете бомб времен холодной войны. Микропроцессорная революция продолжает ускоряться. Темп грядущих изменений превзойдет то, что было в прошедшие десятилетия. Теперь у вас есть инструменты для разработки и построения систем, которые сформируют наше будущее. С этими знаниями приходит и большая ответственность. Мы надеемся, что вы используете их не только для развлечения и обогащения, но и на пользу человечеству.

Упражнения

Упражнение 8.1 В пределах одной страницы текста опишите четыре повседневных занятия, обладающих временной или пространственной локальностью. Приведите два конкретных примера для каждого типа локальности.

Упражнение 8.2 Одним абзацем опишите два коротких компьютерных приложения, обладающих временной или пространственной локальностью. Опишите, как именно это происходит.

Упражнение 8.3 Придумайте последовательность адресов, для которых кеш с прямым отображением емкостью 16 слов и длиной страницы, равной четырем словам, будет более производительным, чем полностью ассоциативный кеш с такой же емкостью и длиной строки, использующий стратегию вытеснения редко используемых данных (LRU).

Упражнение 8.4 Повторите **упражнение 8.3** для случая, когда полностью ассоциативный кеш более производителен, чем кеш с прямым отображением.

---

[1] Детектив-персонаж комикса 1930-х годов, у которого был телефон в наручных часах. – *Прим. перев.*

Упражнение 8.5 Опишите компромиссы при увеличении каждого из следующих параметров кеша при сохранении остальных параметров неизменными:

(a) длина строки;
(b) степень ассоциативности;
(c) емкость кеша.

Упражнение 8.6 Доля промахов у двухсекционного ассоциативного кеша всегда меньше, обычно меньше, иногда меньше или никогда не меньше, чем у кеша с прямым отображением с такой же емкостью и длиной строки? Дайте аргументированный ответ.

Упражнение 8.7 Приведенные ниже утверждения относятся к доле промахов кеша. Укажите, истинно ли каждое из утверждений. Кратко объясните ход рассуждений; приведите контрпример, если утверждение ложно.

(a) Доля промахов у двухсекционного ассоциативного кеша всегда ниже, чем у кеша прямого отображения с такой же емкостью и длиной строки.
(b) Доля промахов у 16-килобайтного кеша прямого отображения всегда ниже, чем у 8-килобайтного кеша прямого отображения с той же длиной строки.
(c) Доля промахов у кеша команд с 32-байтной строкой обычно ниже, чем у кеша команд с 8-байтной строкой при той же емкости и степени ассоциативности.

Упражнение 8.8 Дан кеш со следующими параметрами: b, длина строки в словах; S, количество наборов; N, количество секций; A, количество битов адреса.

(a) Выразите через перечисленные параметры емкость кеша C.
(b) Выразите через перечисленные параметры количество битов, необходимое для хранения тегов.
(c) Чему равны S и N для полностью ассоциативного кеша емкостью C слов, длина строки которого равна b?
(d) Чему равно S для кеша прямого отображения кеша емкостью C слов, длина строки которого равна b?

Упражнение 8.9 Дан содержащий 16 слов кеш, параметры которого приведены в **упражнении 8.8**. Рассмотрите следующую повторяющуюся последовательность шестнадцатеричных адресов для команд загрузки (`lw`):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Предполагая использование стратегии вытеснения редко используемых данных (LRU) для ассоциативных кешей, определите доля промахов при выполнении этой последовательности команд при использовании одного из приведенных ниже кешей. Неизбежными промахами (compulsory misses) можно пренебречь.

(a) кеш прямого отображения, $b = 1$ слово;
(b) полностью ассоциативный кеш, $b = 1$ слово;
(c) двухсекционный ассоциативный кеш, $b = 1$ слово;
(d) кеш прямого отображения, $b = 2$ слова.

Упражнение 8.10 Повторите **упражнение 8.9** для следующей повторяющейся последовательности шестнадцатеричных адресов для команд загрузки (`lw`) приведенных ниже конфигураций кеша. Емкость кеша – 16 слов:

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) кеш прямого отображения, $b = 1$ слово;
(b) полностью ассоциативный кеш, $b = 2$ слова;
(c) двухсекционный ассоциативный кеш, $b = 2$ слова;
(d) кеш прямого отображения, $b = 4$ слова.

Упражнение 8.11 Предположим, что выполняется программа со следующей последовательностью адресов обращений к памяти, выполняющейся один раз:

0x0 0x8 0x10 0x18 0x20 0x28

(a) Если используется кеш прямого отображения емкостью 1 Кбайт и длиной строки 8 байт (2 слова), то сколько в кеше наборов?
(b) Какова доля промахов кеша прямого отображения для данной последовательности обращений? Емкость и длина строки такие же, как в пункте (a).
(c) Что сильнее всего сократит долю промахов при данной последовательности обращений к памяти? Емкость кеша постоянна. Выберите один из вариантов.
    (1) Увеличение степени ассоциативности до двух.
    (2) Увеличение длины строки до 16 байт.
    (3) Или (1), или (2).
    (4) Ни (1), ни (2).

Упражнение 8.12 Вы разрабатываете кеш команд для процессора RISC-V. Его емкость равна $4C = 2^{c+2}$ байт. Его степень ассоциативности $N = 2^n$ ($N \geq 8$), длина строки (размер блока) $b = 2^b$ байт ($b \geq 8$). Используя эти сведения, ответьте на следующие вопросы.

(a) Какие биты адреса используются для выбора слова в строке?
(b) Какие биты адреса используются для выбора набора в кеше?
(c) Сколько битов в каждом теге?
(d) Сколько битов тега во всем кеше целиком?

Упражнение 8.13 Дан кеш со следующими параметрами: N (ассоциативность) = 2, b (длина строки) = 2 слова, W (размер слова) = 32 бита, C (емкость кеша) = 32K слов, A (размер адреса) = 32 бита. Нас интересуют только адреса слов.

(a) Какие биты адреса занимают тег, индекс (номер набора), смещение в строке и байтовое смещение? Укажите, сколько битов требуется для каждого из этих полей.
(b) Каков общий размер *всех* тегов кеш-памяти в битах?
(c) Предположим, что каждая строка кеша содержит бит достоверности (V) и бит изменения (D). Чему равен размер набора кеша, включая данные, тег и служебные биты?
(d) Разработайте кеш, используя строительные блоки, показанные на **рис. 8.28**, и небольшое количество двухвходовых логических элементов. Кеш должен включать память тегов, память данных, сравнение адресов, выбор данных, поступающих на выход, а также любые другие элементы, которые вы сочтете необходимыми. Учтите, что мультиплексоры и компараторы могут быть любой ширины (n или p бит соответственно), но блоки памяти SRAM должны обязательно быть 16K × 4 бит. Не забудьте приложить блок-схему с пояснениями. Вам достаточно реализовать только операции чтения.

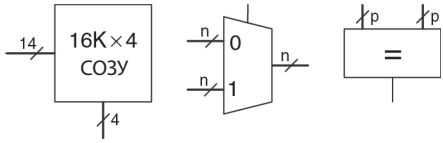


Рис. 8.28 Строительные блоки

Упражнение 8.14 Вас приняли на работу в новый перспективный интернет-стартап, разрабатывающий наручные часы со встроенным пейджером и веб-браузером. В разработке используется встроенный процессор с многоуровневой схемой кеширования, изображенной на **рис. 8.29**. Процессор включает небольшой кеш на чипе и большой внешний кеш второго уровня (да, часы весят полтора килограмма, но зато отлично подходят для серфинга в интернете!).

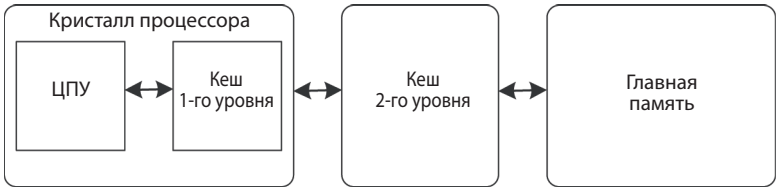


Рис. 8.29 Компьютерная система

Предположим, что процессор использует 32-битные физические адреса, но обращается к данным только по адресам, выровненным по границе слова. Характеристики кешей приведены в **табл. 8.4**. Время доступа к оперативной памяти (main memory) равно t_m, а ее размер – 512 Мбайт.

Таблица 8.4 Характеристики памяти

Характеристика	Кеш на чипе	Внешний кеш
Способ организации	4-секционный наборно-ассоциативный	Прямого отображения
Доля попаданий	A	B
Время доступа	t_a	t_b
Длина строки	16 байт	16 байт
Количество строк	512	256K

(a) Во сколько разных мест в кеше на чипе и в кеше второго уровня может быть загружено слово, находящееся в памяти?

(b) Чему равен размер тега в битах для кеша на чипе и кеша второго уровня?

(c) Напишите выражение для вычисления среднего времени доступа к данным при чтении. Обращение к кешам происходит последовательно[1].

(d) Измерения показывают, что для некоторой задачи доля попаданий в кеш на чипе – 85 %, а в кеш второго уровня – 90 %. Но если кеш

[1] То есть сначала к кешу на чипе, а затем к кешу второго уровня, после чего – к оперативной памяти. – *Прим. перев.*

на чипе отключен, то доля попаданий в кеш второго уровня достигает 98,5 %. Дайте краткое объяснение такому поведению.

Упражнение 8.15 В этой главе был описан механизм замещения редко используемых строк (LRU) для многосекционных наборно-ассоциативных кешей. Существуют и другие, хотя и менее распространенные, механизмы замещения, такие как «первым пришел, первым ушел» (first-in-first-out, FIFO) и замещение в случайном порядке. Замещение по принципу FIFO вытесняет ту строку, которая находится в кеше дольше всего, независимо от того, как давно к ней обращались. Случайное замещение выбирает строку для вытеснения случайным образом.

(a) Каковы достоинства и недостатки каждого из этих механизмов?
(b) Опишите последовательность обращений к памяти, при которой FIFO будет работать лучше, чем LRU.

Упражнение 8.16 Вы разрабатываете вычислительную систему с иерархической системой памяти, состоящей из раздельного кеша команд и данных и оперативной памяти, и используете многотактный процессор RISC-V, показанный на рис. 7.44, работающий на частоте 1 ГГц.

(a) Предположим, что кеш команд работает идеально (т. е. промахов кеша нет), а доля промахов кеша данных – 5 %. При промахе кеша процессор приостанавливает выполнение текущей команды на 60 нс, в течение которых происходит доступ к оперативной памяти, после чего возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кеша?
(b) Сколько тактов на команду (clocks per instruction, CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?
(c) Рассмотрим тестовое приложение из примера 7.7, содержащее 25 % команд загрузки, 10 % команд сохранения, 11 % команд условных переходов (ветвлений), 2 % команд безусловных переходов и 52 % команд типа R[1]. Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?
(d) Теперь предположим, что кеш команд тоже неидеален и его промахи происходят в 7 % случаев. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи и кеша команд, и кеша данных.

Упражнение 8.17 Повторите **упражнение 8.16** со следующими параметрами:

(a) Кеш команд идеален (т. е. промахов кеша нет), а доля промахов кеша данных – 15 %. При промахе процессор приостанавливает выполнение текущей команды на 200 нс, в течение которых обращается к оперативной памяти, а затем возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кеша?
(b) Сколько тактов на команду (clocks per instruction, CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?

[1] Данные из книги Patterson and Hennessy, Computer Organization and Design, 4th Edition, Morgan Kaufmann, 2011. (Использовано с разрешения авторов.)

(c) Рассмотрим тестовое приложение из **примера 7.7**, содержащее 25 % команд загрузки, 10 % команд сохранения, 11 % команд условных переходов (ветвлений), 2 % команд безусловных переходов и 52 % команд типа R. Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?

(d) Теперь предположим, что кеш команд тоже неидеален и его промахи происходят в 10 % случаев. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи и кеша команд, и кеша данных.

Упражнение 8.18 Если в компьютере используются 64-битные виртуальные адреса, сколько виртуальной памяти он может адресовать? Помните, что 2^{40} байт = 1 терабайт, 2^{50} байт = 1 петабайт, а 2^{60} байт = 1 эксабайт.

Упражнение 8.19 Разработчик суперкомпьютера решил потратить 1 млн долларов на оперативную память, изготовленную по технологии DRAM, и столько же на жесткие диски для виртуальной памяти. Сколько физической и виртуальной памяти будет у этого компьютера при ценах, представленных на **рис. 8.4**? Какого размера должны быть физические и виртуальные адреса для обращения к этой памяти?

Упражнение 8.20 Рассмотрим подсистему виртуальной памяти, способную адресовать в общей сложности 2^{32} байт. У вас есть неограниченное дисковое пространство, но всего 8 Мбайт полупроводниковой (физической) памяти. Предположим, что размер физических и виртуальных страниц равен 4 Кбайт.

(a) Чему будет равна длина физического адреса в битах?

(b) Каково максимальное количество виртуальных страниц в данной подсистеме?

(c) Сколько физических страниц в этой подсистеме?

(d) Каков размер номеров виртуальных и физических страниц в битах?

(e) Допустим, вы решили остановиться на схеме прямого отображения виртуальных страниц на физические. При таком отображении для определения номера физической страницы используются младшие биты номера виртуальной страницы. Сколько виртуальных страниц отображается на каждую физическую страницу в этом случае? Почему такое отображение – плохая идея?

(f) Очевидно, что необходима более гибкая и динамичная схема трансляции виртуальных адресов в физические, нежели та, что была описана в пункте (e). Предположим, что вы используете таблицу страниц для хранения отображений (то есть информации, позволяющей однозначно соотнести виртуальную страницу с физической). Сколько элементов будет в такой таблице?

(g) Предположим, что каждая запись в таблице страниц содержит, помимо номера физической страницы, еще и некоторую служебную информацию, состоящую из битов достоверности (V) и изменения (D). Сколько байтов понадобится для хранения каждой записи? Округлите результат вверх до ближайшего целого количества байтов.

(h) Нарисуйте эскиз таблицы страниц. Каков общий размер таблицы в байтах?

Упражнение 8.21 Рассмотрим подсистему виртуальной памяти, способную адресовать 2^{50} байт. У вас есть неограниченное дисковое пространство, но всего

2 Гбайт полупроводниковой (физической) памяти. Предположим, что размер физической и виртуальной страниц равен 4 Кб.

(a) Чему будет равна длина физического адреса в битах?
(b) Каково максимальное количество виртуальных страниц в подсистеме?
(c) Сколько физических страниц в подсистеме?
(d) Каков размер номеров виртуальных и физических страниц в битах?
(e) Сколько записей будет в таблице страниц?
(f) Предположим, что каждая запись в таблице страниц содержит, помимо номера физической страницы, еще и некоторую служебную информацию, состоящую из битов достоверности (V) и изменения (D). Сколько байтов понадобится для хранения каждой записи? Округлите результат вверх до ближайшего целого количества байтов.
(g) Нарисуйте эскиз таблицы страниц. Каков общий размер таблицы в байтах?

Упражнение 8.22 Вы решили ускорить работу подсистемы виртуальной памяти, описанной в **упражнении 8.20**, при помощи буфера ассоциативной трансляции (TLB). Предположим, что система памяти обладает характеристиками, приведенными в **табл. 8.5**. Доля промахов TLB и кеша показывает, как часто требуемый элемент не будет найден в соответствующей памяти. Доля промахов оперативной памяти показывает, как часто случаются страничные ошибки.

Таблица 8.5 Характеристики памяти

Тип памяти	Время доступа в тактах	Процент промахов
TLB	1	0,05 %
Кеш	1	2 %
Оперативная память	100	0,0003 %
Жесткий диск	1 000 000	0 %

(a) Каково среднее время доступа в подсистеме виртуальной памяти до и после добавления TLB? Считайте, что таблица страниц всегда расположена в физической памяти и никогда не загружается в кеш данных.
(b) Чему равен суммарный размер TLB, состоящего из 64 элементов, в битах? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к своим результатам.
(c) Сделайте эскиз TLB. Обозначьте все поля и их размеры.
(d) SRAM какого размера потребуется для организации TLB, описанного в пункте (c)? Дайте ответ в форме «глубина × ширина».

Упражнение 8.23 Вы решили ускорить систему виртуальной памяти из **упражнения 8.21** с помощью буфера трансляции адресов (TLB) из 128 элементов.

(a) Каково суммарное количество битов в TLB? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к этим результатам.
(b) Нарисуйте эскиз TLB. Обозначьте все поля и их размеры.

(c) Блок памяти SRAM какого размера потребуется для организации описанного в пункте (b) TLB? Ответ должен выглядеть как «глубина × ширина».

Упражнение 8.24 Предположим, что многотактный процессор RISC-V, описанный в **разделе 7.4**, использует подсистему виртуальной памяти.

(a) Покажите расположение TLB в блок-схеме многотактного процессора.
(b) Опишите, как добавление TLB повлияет на производительность процессора.

Упражнение 8.25 Подсистема виртуальной памяти, которую вы разрабатываете, использует реализованную аппаратно одноуровневую таблицу страниц, включающую блоки памяти SRAM и сопутствующую логику. Она поддерживает 25-битные виртуальные адреса, 22-битные физические адреса и страницы размером 2^{16} байт (64 Кбайт). В каждой записи таблицы страниц имеется бит достоверности V и бит изменения D.

(a) Чему равен размер всей таблицы в битах?
(b) Группа разработчиков операционной системы предлагает сократить размер страницы с 64 до 16 Кбайт, но разработчики аппаратуры возражают, мотивируя это стоимостью дополнительной аппаратуры. Объясните, почему они против.
(c) Таблица страниц будет расположена рядом с кеш-памятью на том же чипе, что и процессор. Эта кеш-память работает только с физическими (а не виртуальными) адресами. Возможно ли при доступе в память одновременно обращаться к нужному набору в кеше и к таблице страниц? Кратко объясните условия, необходимые для одновременного обращения к набору кеша и к записи таблицы страниц.
(d) Возможно ли при доступе в память одновременно выполнять сравнение тегов и обращаться к таблице страниц? Дайте краткие объяснения.

Упражнение 8.26 Опишите ситуацию, при которой наличие виртуальной памяти могло бы повлиять на разработку приложения. Порассуждайте о том, как размер страницы и физической памяти влияют на производительность приложения.

Упражнение 8.27 Предположим, что у вас есть персональный компьютер (ПК), использующий 32-битные виртуальные адреса.

(a) Чему равен максимальный объем виртуальной памяти, доступный каждой программе?
(b) Как влияет на производительность размер жесткого диска ПК?
(c) Как влияет на производительность размер физической памяти ПК?

Вопросы для собеседования

Следующие упражнения задавались в качестве вопросов на собеседованиях при приеме на работу.

Вопрос 8.1 Объясните разницу между кешем прямого отображения, наборно-ассоциативным и полностью ассоциативным кешами. Для каждого типа кеша

опишите приложение, для которого кеш данного типа будет более эффективен, чем остальные.

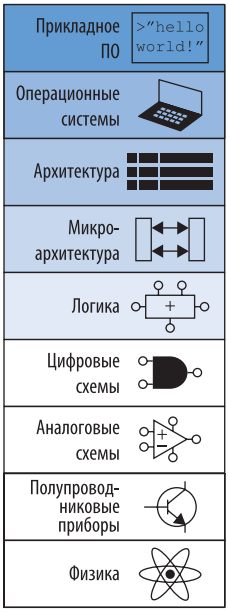
Вопрос 8.2 Объясните, как работают подсистемы виртуальной памяти.

Вопрос 8.3 Объясните достоинства и недостатки использования подсистем виртуальной памяти.

Вопрос 8.4 Объясните, как производительность кеша может зависеть от размера виртуальной страницы в подсистеме виртуальной памяти.

ГЛАВА 9

Ввод/вывод во встраиваемых системах



9.1. Введение

Системы ввода/вывода (Input/Output, I/O) применяются для подключения компьютера к внешним устройствам, которые называются периферийными. В персональном компьютере периферийные устройства включают в себя клавиатуру, монитор, принтер и беспроводную сеть. Во встраиваемых системах вместо них могут быть подключены нагревательный элемент тостера, синтезатор речи куклы, топливный инжектор двигателя, двигатель позиционирования солнечной панели спутника и т. д. Процессор получает доступ к устройствам ввода/вывода, используя шины адреса и данных так же, как при получении доступа к памяти.

В этой главе мы рассмотрим конкретные примеры устройств ввода/вывода. В **разделе 9.2** описаны основные принципы взаимодействия устройства ввода/вывода с процессором и доступа к нему из программы. В **разделе 9.3** ввод/вывод рассматривается в контексте встраиваемых систем. В нем показано, как использовать плату RED-V RedBoard производства компании SparkFun, в которой установлен микроконтроллер RISC-V, для доступа к встроенным периферийным устройствам, включая универсальный, последовательный и аналоговый ввод/вывод, а также таймеры и широтно-импульсную модуляцию (PWM). В **разделе 9.4** приведены примеры взаимодействия с другими распространенными периферийными устройствами, такими как символьные ЖК-дисплеи, мониторы VGA, радиомодули Bluetooth и двигатели.

9.2. Отображение ввода/вывода в пространство памяти

Как было уже сказано в **разделе 6.6.1**, часть адресного пространства вместо памяти отводится под устройства ввода/вывода. Например, допустим, что адреса в диапазоне от 0x20000000 до 0x20FFFFFF используются для доступа к устройствам ввода/вывода. Каждому устройству ввода/вывода присваивается один или несколько адресов в этом диапазоне. При записи по заданному адресу данные отправляются в устройство. При чтении данные поступают из устройства. Этот метод связи с устройствами ввода/вывода называется вводом-выводом, *отображенным в память* (memory-mapped I/O, MMIO).

Встраиваемые процессоры названы так потому, что они обычно встроены в более крупную систему (например, игрушку или автомобиль) и имеют ограниченный пользовательский интерфейс. Напротив, процессоры, используемые в ПК, имеют такие интерфейсы, как клавиатуры и экраны, позволяющие нам заниматься программированием или запускать приложения. Но все типы процессоров по сути одинаковы — все они выполняют инструкции. Различаются только интерфейсы и периферийные устройства, используемые встраиваемыми и традиционными процессорами.

В системе с MMIO процедуры загрузки и сохранения могут осуществлять доступ или к памяти, или к устройству ввода/вывода. На **рис. 9.1** изображены аппаратные средства, необходимые для поддержки двух устройств ввода/вывода, отображенных в память. *Дешифратор адреса* определяет, какое устройство нуждается в соединении с процессором. Он использует сигналы *Address* и *MemWrite*, чтобы генерировать управляющие сигналы для остальной части аппаратных средств. Мультиплексор *ReadData* выполняет переключение между памятью и различными устройствами ввода/вывода. Регистры с разрешением записи хранят значения, передаваемые в устройства ввода/вывода.

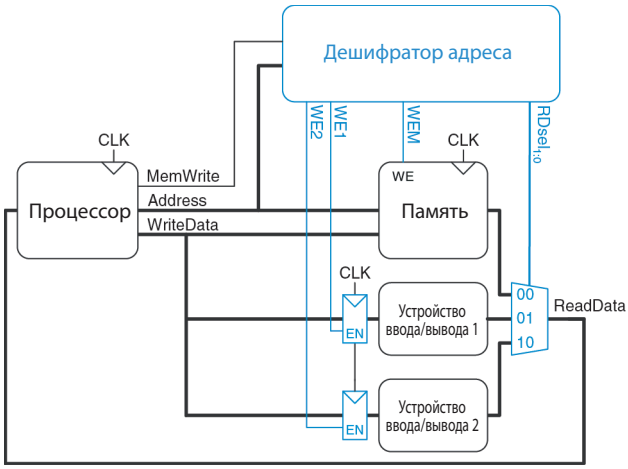


Рис. 9.1 Отображение устройств ввода/вывода в память

Пример 8.17 ОБМЕН ДАННЫМИ С УСТРОЙСТВАМИ ВВОДА/ВЫВОДА

Допустим, что устройству ввода/вывода 1 на **рис. 9.1** назначен адрес памяти 0x20001000. Разработайте код на языке ассемблера для RISC-V, предназначенный для записи значения 7 в устройства ввода/вывода 1 и для чтения данных из устройства ввода/вывода 1.

Решение Приведенный ниже ассемблерный код для RISC-V записывает значение 7 в устройство ввода/вывода 1. Директива ассемблера `.equ` заменяет именованный символ заданным значением. Следовательно, инструкции `li s1, ioadr` соответствует инструкция `li s1, 0x20001000`.

```
.equ ioadr 0x20001000
     li s0, 7
     li s1, ioadr
     sw s0, 0(s1)
```

Дешифратор адреса устанавливает сигнал *WE*1, потому что адрес равен 0x20001000 и сигнал *MemWrite* равен единице. Данные на шине *WriteData* (значение 7) записываются в регистр, который подключен ко входу устройства ввода/вывода 1.

Чтобы считать данные из устройства ввода/вывода 1, процессор должен выполнить инструкцию

```
lw s0, 0(s1)
```

Дешифратор адреса устанавливает $RDsel_{1:0}$ в 01, поскольку он определяет, что адрес – 0x20001000 и значение *MemWrite* равно нулю. Выходные данные *RData1* из устройства ввода/вывода 1 проходят через мультиплексор на шину *ReadData* и загружаются в регистр процессора `s0`.

Адреса, связанные с устройствами ввода/вывода, часто называют *регистрами ввода/вывода*, поскольку они на самом деле указывают на физические регистры в устройстве ввода/вывода, как показано на **рис. 9.1**.

Программное обеспечение, которое взаимодействует с устройством ввода/вывода, называется *драйвером устройства*. Вы, вероятно, загружали или устанавливали драйверы для вашего принтера или другого устройства ввода/вывода. Разработка драйвера требует детального знания аппаратной части этого устройства, включая знание адресов и принципа работы соответствующих регистров. Другие программы вызывают функции в драйвере для получения доступа к устройству без необходимости понимания низкоуровневого аппаратного обеспечения.

9.3. Ввод/вывод во встраиваемых системах

Встраиваемые системы используют процессор для управления взаимодействием с физической средой. Обычно они выстроены вокруг микроконтроллеров (МК), которые сочетают в себе микропроцессор с набором простых в использовании периферийных устройств, таких как цифровые и аналоговые порты ввода/вывода общего назначения, последовательные порты, таймеры и т. д. В большинстве своем МК недорогие и спроектированы так, чтобы минимизировать стоимость и размеры систем путем интеграции большей части необходимых компонентов в один чип. Большинство из них меньше и легче, чем монета в десять центов, потребляют милливатты мощности, и цена их варьируется в пределах от нескольких десятков центов до нескольких долларов. Микроконтроллеры классифицируются по разрядности данных, с которыми они оперируют. 8-битные микроконтроллеры — самые маленькие и самые дешевые, в то время как 32-битные микроконтроллеры обладают большей памятью и имеют большую производительность.

По данным IC Insights, в 2020 году было продано около 24 млрд микроконтроллеров, и прогнозируется, что рынок будет расти со скоростью 10 % в год до 2029 года. Средняя цена отдельного микроконтроллера составляет около 60 центов, а интеграция 8-битного микроконтроллера в систему на кристалле (SoC) обходится в несколько центов. Микроконтроллеры стали использоваться повсеместно и почти незаметны, в 2021 году в среднестатическом новом автомобиле установлено около 100 или более микроконтроллеров.
Автомобильная промышленность — самый крупный и быстрорастущий потребитель микроконтроллеров, за ним следуют бытовая электроника, промышленные системы, медицинские устройства и военная отрасль. По итогам 2020 года наибольший доход принесли 16-разрядные микроконтроллеры, но доля рынка 32-разрядных микроконтроллеров растет из-за их более широких возможностей. Ведущими производителями микроконтроллеров являются Infineon, Microchip, NXP, Renesas, STMicroelectronics и Texas Instruments. Ведущие архитектуры включают 8051, AVR, PIC и ARM. Архитектура ARM — практически монополист в качестве процессора приложений для 90 % мобильных устройств. Несмотря на это, RISC-V вызывает большой интерес как новая архитектура с открытым исходным кодом.

9.3.1. Плата RED-V

RED-V RedBoard в этой главе называется просто RED-V.

Для большей конкретики в этом разделе мы рассмотрим организацию ввода/вывода во встраиваемой системе на примере реального устройства. В частности, мы сосредоточимся на *системе на кристалле* (SoC) FE310-G002 от SiFive, которая содержит 32-разрядный процессор RISC-V с тактовой частотой 320 МГц, реализующий архитектуру RV32IMAC, то есть базовый 32-разрядный целочисленный набор команд (RV32I) плюс расширения умножения/де-

Дополнительные материалы к этой книге, упомянутые в предисловии, включают лабораторные упражнения с использованием платы RED-V.

ления (M), атомарные операции с памятью (A) и сжатые 16-битные инструкции (C). Этот микроконтроллер можно приобрести на отладочной плате HiFive от SiFive, а также на некоторых отладочных платах сторонних производителей, таких как серия RED-V от SparkFun (доступны как в формате Arduino, так и Thing Plus). За описаниями интерфейсов ввода/вывода, приведенными в каждом подразделе, будут следовать конкретные примеры, которые работают на FE310. Все примеры были проверены на плате RED-V RedBoard от SparkFun и могут быть легко запущены на плате HiFive или адаптированы к плате RED-V Thing Plus.

На **рис. 9.2 (a)** представлена плата SparkFun RED-V RedBoard, которую можно приобрести менее чем за 40 долларов. Размеры этой платы 68×53 мм. На рисунке также показаны названия сигналов каждого вывода, которые мы описываем в этом разделе. Отладочная плата может питаться от источника питания USB 5 В или от источника постоянного тока с напряжением от 7 до 15 В через круглый штекерный разъем. Процессор FE310-G002 питается от встроенных стабилизаторов напряжения 3,3 В и 1,8 В. FE310-G002 имеет кеш инструкций L1 размером 16 Кбайт и регистровую память для хранения данных SRAM объемом 16 Кбайт. На отладочной плате SparkFun также установлена флеш-память объемом 32 Мбайта, доступная через последовательный интерфейс (SPI), которую можно использовать для хранения программ и данных.

На **рис. 9.2 (b)** представлена плата RED-V Thing Plus, которая обладает возможностями, аналогичными RED-V RedBoard, но в меньшем форм-факторе (58×23 мм). Ее можно установить на макетную плату для облегчения монтажа схемы. Контакты интерфейса ввода/вывода пронумерованы иначе, чем на RedBoard, и их трудно прочитать на поверхности чипа, но они показаны на рисунке.

Внимание: попадание напряжения 5 В на один из входов/выходов процессора с рабочим напряжением 3,3 В может привести к повреждению этого входа/выхода и, возможно, к выходу из строя всего процессора FE310. Если вы измеряете напряжение на контактах платы с помощью вольтметра, будьте осторожны, чтобы случайно не соединить щупом контакты VUSB или VBAT с соседними контактами!

Форм-фактор RED-V RedBoard целенаправленно скопирован с платы Arduino R3, чтобы сохранить максимальную совместимость со многими расширениями (шилдами) Arduino, разработанными для этой платы. Все 19 настраиваемых каналов ввода/вывода доступны через контакты разъема платы и работают с логическими уровнями 3,3 В. На разъем также выведены линии питания 3,3 В, 5 В и GND (земля) для удобства питания небольших устройств, подключенных к RedBoard, но максимальный отдаваемый ток составляет 50 мА от источника 3,3 В и – 300 мА от источника 5 В.

Стремление поддерживать совместимость с платой Arduino R3 привело к появлению нескольких имен для каждого контакта: подписи к выводам соответствуют стандартным номерам контактов Arduino, но нумерация выводов RED-V, представленная на **рис. 9.2**, отражает как номера контактов Arduino, так и соответствую-

щие номера контактов GPIO (универсальный интерфейс ввода/вывода) процессора FE310. Например, как показано на **рис. 9.2**, контакту GPIO5 (вывод 5 FE310) соответствует D13 (вывод 13 Arduino). На плате RED-V Thing Plus не указаны названия контактов, совместимые с Arduino, но это позволяет избежать наличия нескольких номеров для одного контакта. Также обратите внимание, что некоторые контакты GPIO имеют несколько предназначений. Например, GPIO18 (D2) еще может работать как линия передачи для UART 1 (UART1_TX), как будет описано позже.

На обеих платах вывод GPIO5 подключен к синему светодиоду. Этот вывод имеет маркировку 13 (то есть D13) на RedBoard и 5 на плате Thing Plus.

Данный раздел начинается с описания SoC FE310-G002 и описания общего драйвера устройства для интерфейса ввода/вывода с отображением в память. В оставшейся части этого раздела показано, как встраиваемые системы реализуют универсальный цифровой, аналоговый и последовательный ввод/вывод.

Компания SiFive была основана в 2015 году тремя исследователями из Калифорнийского университета в Беркли: Крсте Асановичем, Юнсупом Ли и Эндрю Уотерманом. Предназначение SiFive — сделать разработку нестандартных микросхем более быстрой и доступной. Опираясь на открытую архитектуру набора инструкций RISC-V (instruction set architecture, ISA), они разработали платформу, которая позволяет проектировать собственные микросхемы на уровне системы. Более подробную информацию, включая техническое описание FE310-G002, можно найти на сайте http://sifive.com.

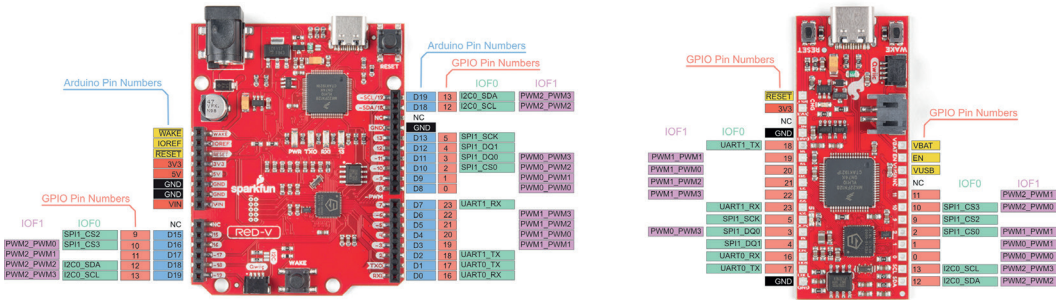


Рис. 9.2 Плата RED-V RedBoard (a) и RED-V Thing Plus (b)
(фотографии предоставлены SparkFun по лицензии CC BY 2.0)

9.3.2. Система на кристалле FE310-G002

Система на кристалле FE310-G002 — это мощный, но недорогой микроконтроллерный чип, разработанный SiFive. Он содержит микропроцессор RISC-V с пятистадийным конвейером, подобный тому, который описан в **главе 7**, и множество периферийных устройств ввода/вывода. FE310 помещен в квадратный корпус с 48 планарными выводами. В документации SiFive есть таблица, в которой описаны многие функции и регистры ввода/вывода; в данной главе мы рассмотрим только часть этих функций.

В **табл. 9.1** представлена карта памяти FE310. После сброса процессор начинает выполнение кода из внешней флеш-памяти по адресу 0x20000000. В документации предусмотрено распределение адресов флеш-памяти объемом до 512 Мбайт, хотя на современных платах RED-V памяти гораздо меньше: RED-V RedBoard обладает 32 Мбайтами внешней флеш-памяти, а RED-V Thing Plus – 4 Мбайтами. Чип процессора также имеет 16 Кбайт ОЗУ, так называемой *плотно интегрированной памяти данных* (data tightly integrated memory, DTIM), по адресу 0x80000000. Эта оперативная память имеет задержку загрузки в 2 такта и используется для хранения переменных. Различные периферийные устройства отображаются в память между адресами 0x02000000 и 0x1FFFFFFF. Они будут подробно описаны в следующих разделах. Эти периферийные устройства включают каналы *универсального ввода/вывода* (GPIO), три блока *широтно-импульсной модуляции* (ШИМ) для генерации выходных сигналов и различные последовательные порты для подключения к внешним устройствам, включая три *последовательных периферийных интерфейса* (serial peripheral interfaces, SPI), два *универсальных асинхронных приемника/передатчика* (universal asynchronous receiver/transmitter, UART) и один I^2C *интерфейс* (inter-integrated circuit).

На **рис. 9.3** показана упрощенная схема платы RED-V RedBoard. Плата получает питание 5 В от внешнего источника через разъем USB, а регуляторы формируют напряжения 3,3 В и 1,8 В для интерфейса ввода/вывода, питания постоянно включенного ядра с низким энергопотреблением и прочих нужд.

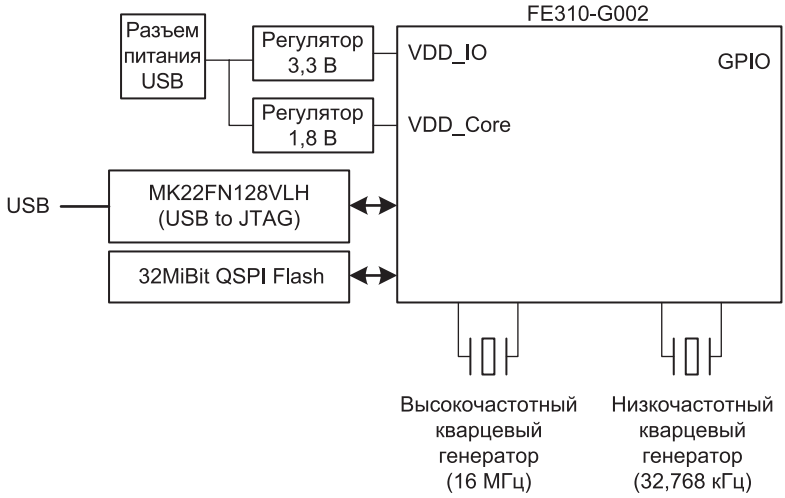


Рис. 9.3 Упрощенная схема платы RED-V

Таблица 9.1 Карта памяти процессора FE310

Нижний адрес	Верхний адрес	Атрибут	Описание	Примечание
0X0000_0000	0x0000_0FFF	RWX A	Отладка	Отладочное пространство
0X0000_1000	0X0000_1FFF	R XC	Выбор режима	Внутренняя энергонезависимая память
0X0000_2000	0X0000_2FFF		Зарезервировано	
0x6006_3000	0x0000_3FFF	RWX A	Ошибочное устройство	
0X0000_4000	0X0000_FFFF		Зарезервировано	
0X0001_0000	0X0001_1FFF	R XC	Масочное ПЗУ (8 Кбайт)	
0X0001_2000	0x0001_FFFF		Зарезервировано	
0X0002_0000	0X0002_1FFF	R XC	Область памяти OTP	
0X0002_2000	0X001F_FFFF		Зарезервировано	
0X0200_0000	0X0200_FFFF	RW A	CLINT	Внутренние периферийные устройства
0X0201_0000	0X07FF_FFFF		Зарезервировано	
0X0800_0000	0X0800_1FFF	RWX A	E31 ITIM (8 Кбайт)	
0X0800_2000	0X0BFF_FFFF		Зарезервировано	
0X0C00_0000	0X0FFF_FFFF	RW A	PLIC	
0X1000_0000	0x1000_0FFF	RW A	AON	
0X1000_1000	0X1000_7FFF		Зарезервировано	
0X1000_8000	0X1000_8FFF	RW A	PRCI	
0X1000_9000	0x1000_FFFF		Зарезервировано	
0X1001_0000	0X1001_0FFF	RW A	Управление OTP	
0X1001_1000	0X1001_1FFF		Зарезервировано	
0X1001_2000	0x1001_2FFF	RW A	GPIO	
0X1001_3000	0x1001_3FFF	RW A	UART 0	
0X1001_4000	0X1001_4FFF	RW A	QSPI 0	
0X1001_5000	0X1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0X1001_7000	0X1002_2FFF		Зарезервировано	
0X1002_3000	0X1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0X1002_6000	0X1003_3FFF		Зарезервировано	
0X1003_4000	0X1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0X1003_6000	0X1FFF_FFFF		Зарезервировано	
0x2000_0000	0X3FFF_FFFF	R XC	QSPI 0 флеш (512 Мбайт)	Внешняя энергонезависимая память
0x4000_0000	0x7FFF_FFFF		Зарезервировано	
0X8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 Кбайт)	Энергозависимая внутренняя память
0x8000_4000	0XFFFF_FFFF		Зарезервировано	

Атрибуты памяти: R – чтение, W – запись, X – выполнение, C – кеширование, A – атомарность. Перепечатано из табл. 4 руководства FE310-G0002, с разрешения © SiFive, Inc., 2019.

Микроконтроллеры RISC-V от компании SiFive продолжают развиваться. К тому времени, как вы будете читать эту книгу, может появиться более новая модель с более продвинутым процессором и другим набором встроенных модулей ввода/вывода. Тем не менее все рассмотренные здесь принципы будут применимы к этому и другим микроконтроллерам. В новом микроконтроллере будут присутствовать аналогичные каналы ввода/вывода и периферийные устройства. Вам лишь потребуется свериться с описанием микроконтроллера, чтобы установить соответствие между периферийным устройством, выводом на микросхеме и выводом на плате, а также найти отображенные в памяти адреса ввода/вывода (регистры), связанные с каждым периферийным устройством. Ваша программа по-прежнему будет выполнять запись в регистры конфигурации для инициализации периферийного устройства, а также читать и записывать содержимое регистров данных для связи с периферийным устройством, как описано в этой главе.

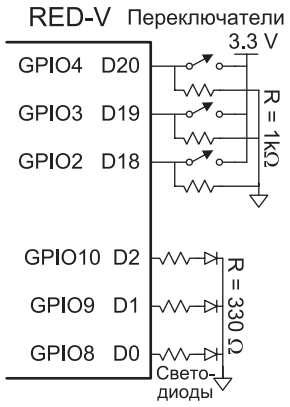


Рис. 9.4 Пример использования выводов GPIO

9.3.3. Цифровой ввод/вывод общего назначения

Порты ввода/вывода общего назначения используются для чтения или записи цифровых сигналов. Как минимум выводам GPIO требуются отображенные в память регистры ввода/вывода для чтения логических уровней на входах порта, записи значений в выходы и установки направления вывода порта (вход или выход). Во многих встраиваемых системах отдельные выводы GPIO могут использоваться совместно с одним или несколькими периферийными устройствами специального назначения, поэтому необходимы дополнительные *регистры конфигурации*, от содержимого которых зависит, является ли вывод порта специализированным. Кроме того, процессор может прервать выполнение программы и обработать прерывание, когда на входном выводе возникает определенное событие, например нарастающий или спадающий фронт, а соответствующее условие прерывания указано в регистре конфигурации. Напомним, что у FE310 есть 19 контактов GPIO. Мы начнем этот раздел с базового примера управления выводами, а затем рассмотрим некоторые специальные назначения данных выводов.

На **рис. 9.4** изображены три светодиода и три переключателя (switches), подключенные к шести контактам GPIO. Светодиоды подключены так, чтобы они светились, когда на выводе присутствует логическая единица, и гасли при логическом нуле. Последовательно со светодиодами включены токоограничивающие резисторы (обычно около 300 Ом), чтобы ограничить яркость светодиодов и избежать перегрузки выводов GPIO. Переключатели подключены таким образом, чтобы при замыкании контактов на выводе GPIO присутствовал высокий логический уровень, а при размыкании – низкий. Как следует из схемы, понижающие резисторы 1 кОм подтягивают выводы к низкому уровню, когда контакты разомкнуты. На **рис. 9.4** показаны как номера контактов Arduino, которые обозначены на плате, так и номера контактов GPIO.

В **табл. 9.2** перечислены регистры GPIO и их смещения адресов относительно базового адреса GPIO 0x10012000, как показано в табл. 51 технического опи-

сания FE310-G002. Давайте сначала сосредоточимся на четырех верхних регистрах (т. е. на отображенных в память адресах ввода/вывода). Каждый вывод GPIO отображается на один разряд регистра. Операция чтения из регистра `input_val` (входные значения) считывает значения (логические уровни) со входов GPIO, а операция записи в регистр `output_val` (выходные значения) приводит к появлению соответствующих логических уровней на выходах GPIO. Перед чтением или записью данных необходимо в регистрах разрешения входов и выходов (`input_en` и `output_en`) настроить линии порта GPIO как входы и выходы, а регистр включения аппаратных функций ввода/вывода (`iof_en`) следует обнулить, чтобы выводы работали в качестве GPIO.

Таблица 9.2 Смещения регистров GPIO

Смещение	Имя	Описание
0x00	input_val	Входное значение на выводе
0x04	input_en	Вывод в режиме входа*
0x08	output_en	Вывод в режиме выхода*
0x0C	output_val	Выходное значение на выводе
0x10	pue	Включение встроенных нагрузочных резисторов*
0x14	ds	Мощность (предельный ток) вывода
0x18	rise_ie	Разрешение прерывания по фронту импульса
0x1C	rise_ip	Запрос прерывания по переднему фронту импульса
0x20	fall_ie	Разрешение прерывания по заднему фронту импульса
0x24	fall_ip	Запрос прерывания по заднему фронту импульса
0x28	high_ie	Разрешение прерывания по высокому уровню сигнала
0x2C	high_ip	Запрос прерывания по высокому уровню сигнала
0x30	low_ie	Разрешение прерывания по низкому уровню сигнала
0x34	low_ip	Запрос прерывания по низкому уровню сигнала
0x38	iof_en	Включение аппаратных функций
0x3C	iof_sel	Выбор аппаратных функций
0x40	out_xor	Операция XOR с выходами (инверсия)

* Эти регистры асинхронно сбрасываются в 0 при включении, так что выводы GPIO неактивны. Перепечатано из табл. 52 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Операции ввода/вывода с отображением GPIO в память

Мы проиллюстрируем использование выводов GPIO, разработав программу, которая считывает состояние переключателей и управляет светодиодами через порт GPIO. Пять наиболее важных регистров для работы с выводами GPIO – это, как показано в **табл. 9.2**, `input_val`, `input_en`, `output_en`, `output_val` и `iof_en` со смещениями 0x0, 0x4, 0x8, 0xC и 0x38 от базового адреса соответственно. Каждый регистр имеет ширину 32 бита и может управлять работой до 32 выводов GPIO, но физически на этом чипе присутствуют только 19 выводов.

Чтобы прочитать логический уровень на выводе *n* GPIO, программа устанавливает бит *n* регистра `input_en` (разрешение входа), а затем считывает регистр `input_val` (входное значение) и проверяет значение бита *n*. Точно так же для управления значением на выходе *n* GPIO программа устанавливает бит *n* регистра `output_en` (разрешение выхода), а затем записывает нужное значение в бит *n* регистра `output_val` (выходное значение). В обоих случаях бит *n* регистра `iof_en` должен быть сброшен, чтобы гарантировать, что с выводом работает именно контроллер GPIO, а не другой периферийный модуль микросхемы.

В контексте манипуляций с битами «установить» означает записать 1 в соответствующий разряд регистра, а «сбросить» означает записать 0 в этот разряд.

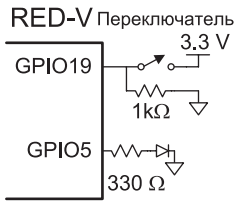


Рис. 9.5 Подключение светодиода к выходу 5 GPIO и кнопки к входу 19 GPIO

В **примере кода 9.1** представлена простая программа, которая считывает состояние переключателя, подключенного к GPIO19, и соответственно включает или выключает светодиод, подключенный к GPIO5. Схема подключения компонентов изображена на **рис. 9.5**. Чтобы получить доступ к отображенному в память порту ввода/вывода, программа сначала объявляет указатели на пять регистров по адресам, указанным в **табл. 9.2**. Каждый указатель имеет тип `uint32_t*`, потому что регистры содержат 32-битные значения без знака. Программа записывает значение 1 в 19-й бит регистра `input_en` и значение 1 в 5-й бит регистра `output_en`, конфигурируя вывод GPIO 19 как цифровой вход и вывод GPIO 5 как цифровой выход. Обратите внимание, как мы используем операцию сдвига (1 << 19) для установки в единицу 19-го бита и выполняем операцию OR с существующим содержимым регистра разрешения, чтобы включить этот бит, не затрагивая другие биты, которые, возможно, уже были включены. Затем мы записываем 0 в биты 5 и 19 в регистре `iof_ en`, чтобы гарантировать, что соответствующие выводы управляются контроллером GPIO. Чтобы записать 0 в нужный бит регистра `iof_en`, мы выполняем над содержимым этого регистра логическую операцию AND с единицами в каждой позиции, кроме этого бита, так что требуемый бит принудительно обнуляется, а другие биты не затрагиваются. Затем программа

многократно считывает логический уровень со входа и записывает его на выход. Чтобы прочитать состояние входа, программа читает регистр `input_val`, сдвигает значение вправо на 19 (чтобы переместить значение бита 19 в разряд 0) и выполняет побитовую операцию AND с числом 0x1, дабы сохранить только значение бита 0, который сейчас хранит значение, изначально находившееся в бите 19. Чтобы записать единицу в бит регистра `output_val`, мы используем операцию OR, как мы это делали для установки бита в регистрах разрешения. Чтобы записать ноль в нужный бит регистра `output_val`, мы используем тот же прием, что и для сброса битов в регистре `iof_en`.

Пример кода 9.1 ОПЕРАЦИИ СО ВХОДАМИ И ВЫХОДАМИ GPIO

```
#include <stdint.h>
int main(void) {
    volatile uint32_t *input_val  = (uint32_t*)0x10012000;
    volatile uint32_t *input_en   = (uint32_t*)0x10012004;
    volatile uint32_t *output_en  = (uint32_t*)0x10012008;
    volatile uint32_t *output_val = (uint32_t*)0x1001200C;
    volatile uint32_t *iof_en     = (uint32_t*)0x10012038;
    int val;

    *iof_en &= ~(1 << 19);                // Вывод 19 управляется GPIO
    *input_en |= (1 << 19);               // Вывод 19 настроен как вход
    *iof_en &= ~ (1 << 5);                // Вывод 5 управляется GPIO
    *output_en |= (1 << 5);               // Вывод 5 настроен как выход
    while (1) {
        val = (*input_val >> 19) & 1;     // Читаем уровень сигнала на входе 19
        if (val) *output_val |= (1 << 5); // Высокий уровень сигнала на выходе 5
        else *output_val &= ~(1 << 5);    // Низкий уровень сигнала на выходе 5
    }
}
```

Другие регистры GPIO

В **табл. 9.2** представлено несколько других заслуживающих внимания регистров управления GPIO, в частности регистры управления мощностью выводов (`ds`), включения встраиваемых подтягивающих резисторов (`pue`) и управления функциями ввода/вывода (`iof_sel` и `iof_en`).

Регистр `ds` задает максимальный выходной ток, протекающий через каждый вывод. Значение по умолчанию (0) настраивает ток в состоянии высокого/низкого логического уровня (IOL/IOH) на уровни 15/16 мА, тогда как установка бита регистра `ds` в единицу увеличивает выходной ток соответствующего вывода до 21 мА, что позволяет напрямую управлять свечением обычного светодиода.

Регистр `pue` управляет подключением внутреннего *подтягивающего* резистора. На **рис. 9.4** показан пример внешнего *понижающего* резистора. Если полярность питания и «земли» в цепи переключателя поменять местами, то резистор из понижающего превратится в подтяги-

вающий и будет формировать на входе высокий уровень, если не замкнуты контакты переключателя. В этом случае при замыкании контактов напряжение на входе будет нулевым. Чтобы сэкономить деньги и место на печатной плате, многие микроконтроллеры содержат внутренние подтягивающие резисторы, которые можно программно подключить к выводу. Запись единицы в бит регистра `pue` активирует внутренний подтягивающий резистор для соответствующего вывода GPIO. Согласно **табл. 4.2** спецификации FE310-G002, когда напряжение на входе равно нулю, ток через подтягивающий резистор составляет 85 мкА. Следовательно, эффективное подтягивающее сопротивление составляет 3,3 В / 85 мкА = 39 кОм (V/I = R).

Таблица 9.3 Специальные функции выводов GPIO

Номер GPIO	IOF0	IOF1
0		PWM0_PWM0
1		PWM0_PWM1
2	SPI1_CS0	PWM0_PWM2
3	SPI1_DQ0	PWM0_PWM3
4	SPI1_DQ1	
5	SPI1_SCK	
6	SPI1_DQ2	
7	SPI1_DQ3	
8	SPI1_CS1	
9	SPI1_CS2	
10	SPI1_CS3	PWM2_PWM0
11		PWM2_PWM1
12	I2C0_SDA	PWM2_PWM2
13	I2C0_SCL	PWM2_PWM3
14		
15		
16	UART0_RX	
17	UART0_TX	
18	UART1_TX	
19		PWM1_PWM1
20		PWM1_PWM0
21		PWM1_PWM2
22		PWM1_PWM3
23	UART1_RX	
24		
25		
26	SPI2_CS0	
27	SPI2_DQ0	
28	SPI2_DQ1	
29	SPI2_SCK	
30	SPI2_DQ2	
31	SPI2_DQ3	

Перепечатано из табл. 53 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Как показано в **табл. 9.3**, большинство выводов GPIO также могут выполнять особые функции, такие как последовательный порт или выход с широтно-импульсной модуляцией (ШИМ). Мы подробно рассмотрим эти функции позже. Регистры `iof_sel` и `iof_en` вместе определяют, работает ли определенный вывод как GPIO или ему назначена специальная функция. Когда бит `iof_en` равен нулю (по умолчанию), соответствующий вывод действует как GPIO. Когда бит равен единице, этот вывод выполняет специальную функцию. Специальная функция выбирается в соответствии с **табл. 9.3** на основе бита `iof_sel` для этого вывода. Например, чтобы использовать вывод GPIO11 для генерации сигнала с широтно-импульсной модуляцией, установите бит 11 регистров `iof_sel` и `iof_en`. Затем используйте регистры ШИМ для управления параметрами сигнала. Регистр `iof_en` отображается на адрес 0x10012038, а регистр `iof_sel` – на 0x1001203C. В **табл. 9.3** перечислены 32 GPIO; но напомним, что платы RED-V поддерживают только 19 GPIO с номерами от 0 до 5, от 9 до 13 и от 16 до 23 включительно.

9.3.4. Драйверы устройств ввода/вывода

Как показано в **примере кода 9.1**, программисты могут управлять устройствами ввода/вывода напрямую, обращаясь для чтения или записи

к отображенным в память регистрам. Но лучше разработать функции, выполняющие эту работу. Такие функции называются *драйверами устройств*. Этот подход имеет множество преимуществ:

- код легче читать, если он содержит вызов функции по имени, а не запись в битовые поля по интуитивно непонятному адресу памяти;
- разработчик, изучивший до мелочей работу устройств ввода/вывода, может разработать драйвер устройства, а обычные пользователи будут использовать его в своих программах, не вникая в детали;
- код легче перенести на другой процессор, отличающийся распределением памяти или устройствами ввода/вывода, поскольку достаточно заменить только нужные драйверы устройств;
- если драйвер устройства является частью операционной системы (ОС), то она может контролировать доступ к физическим устройствам, совместно используемым несколькими программами, запущенными в системе, и управлять безопасностью (например, чтобы вредоносная программа не могла читать с клавиатуры символы пароля, который вы вводите в веб-браузере).

В этой главе мы разработаем простой драйвер устройства под названием EasyREDVIO для доступа к периферийным модулям FE310, чтобы вы лучше разобрались в принципах работы стандартного драйвера. Пользователи, желающие получить доступ ко всем функциям FE310, могут воспользоваться средой разработки Freedom Metal, которая содержит удобные программные интерфейсы для управления функциями SiFive Core IP и периферийными устройствами. Freedom Metal – это мощный инструмент разработчика, поскольку написан таким образом, что его API будет работать на любом устройстве, имеющем пакет поддержки плат Freedom Metal (BSP). BSP – это программный пакет, содержащий драйверы и другие часто используемые процедуры. SiFive также предоставляет комплекты разработчика программного обеспечения (software developers kit, SDK) Freedom E и Freedom Studio, позволяющие пользователям разрабатывать программное обеспечение для любого ядра SiFive.

EasyREDVIO и примеры кода этой главы можно загрузить с веб-сайта поддержки книги, как сказано в предисловии. Более подробную информацию про средства разработки Freedom Metal и документацию можно получить на сайте https://github.com/sifive/freedom-metal.

Пример 9.2 ДРАЙВЕРЫ УСТРОЙСТВА НА ЯЗЫКЕ С

Доступ и изменение значений отображаемых в память регистров ввода/вывода осуществляются путем чтения или записи по соответствующим адресам памяти. На языке ассемблера это делается с помощью инструкций `lw` и `sw`. Как показано в **примере кода 9.2**, на языке С можно сделать то же самое при помощи указателей, но объявлять указатели для каждого отображаемого в память регистра ввода/вывода – утомительный и чреватый ошибками процесс. Более естествен-

ный способ описания и взаимодействия с отображаемыми в память регистрами ввода/вывода в языке C – использование *структур*.

Как сказано в разделе **C8.5 приложения**, структуры в языке C – это способ сгруппировать коллекцию данных разных типов в единый блок. Использование структур в контексте отображаемых в память регистров позволяет обращаться к устройству ввода/вывода с использованием имени данного регистра или поля, а не адреса памяти. Программа на языке C может объявить структуру периферийного устройства, перечисляя регистры в том порядке, в котором они расположены в адресном пространстве памяти. Затем она может объявить *указатель* на такую структуру и получить доступ к периферийному устройству через указатель структуры.

Разработка библиотеки EasyREDVIO будет заключаться в создании кода функций `pinMode`, `digitalRead` и `digitalWrite`, которые позволяют настроить направление вывода и прочитать или записать содержимое регистра.

- Функция `pinMode` ожидает два входных параметра: номер вывода и его режим. Например, `pinMode(5, INPUT)` настраивает вывод GPIO5 как вход, а `pinMode(17, OUTPUT)` устанавливает вывод GPIO17 как выход.
- Функция `digitalRead` ожидает один входной параметр, номер вывода, и возвращает значение логического уровня на этом выводе. Например, `digitalRead(19)` считывает логический уровень на выводе GPIO19.
- Функция `digitalWrite` ожидает два входных параметра: номер вывода и двоичное значение. Например, `digitalWrite(3, 1)` выставляет высокий логический уровень (1) на выводе GPIO3, а `digitalWrite(5, 0)` выставляет низкий логический уровень (0) на выводе GPIO5.

Завершив работу над функциями, разработайте программу на C, которая использует эти функции для чтения состояния трех переключателей и включения соответствующих светодиодов, используя схему, изображенную на **рис. 9.4**.

Решение Код библиотеки EasyREDVIO приведен ниже. Функции должны выбрать, к каким регистрам и битам в этих регистрах обращаться. Например, чтобы настроить вывод как вход, функция `pinMode` должна установить бит этого вывода в регистре `input_en` и сбросить бит этого вывода в регистре `output_en`. Функция `digitalWrite` выполняет запись 1 или 0 в нужный бит регистра `output_val`. Функция `digitalRead` читает нужный бит из `input_val`.

Структура GPIO (`struct`) определяет 32-битные регистры по имени. Затем два оператора `define` определяют базовый адрес GPIO (`GPIO0_BASE`) и создают экземпляр указателя типа GPIO, расположенного по этому базовому адресу. Каждая из 32-битных переменных структуры затем размещается в памяти по возрастанию начиная от этого базового адреса.

Пример кода 9.2 РАБОТА С ПЕРЕКЛЮЧАТЕЛЯМИ И СВЕТОДИОДАМИ

```
// EasyREDVIO.h
// Джошуа Брейк, Дэвид Харрис, Сара Харрис, 7 октября 2020

#include <stdint.h>
```

Пример кода 9.2 (продолжение)

```
// EasyREDVIO.h
// Джошуа Брейк, Дэвид Харрис, Сара Харрис, 7 октября 2020

#include <stdint.h>

#define INPUT 0
#define OUTPUT 1

// Устанавливаем соответствие между номерами выводов Arduino и номерами
// GPIO FE310 в соответствии с рис. 9.2
#define D0 16
#define D1 17
#define D2 18
#define D3 19
#define D4 20
#define D5 21
#define D6 22
#define D7 23
#define D8 0
#define D9 1
#define D10 2
#define D11 3
#define D12 4
#define D13 5
#define D15 9
#define D16 10
#define D17 11
#define D18 12
#define D19 13

// Объявляем структуру GPIO, определяя регистры GPIO в соответствии с порядком их размещения в табл 9.2
typedef struct {
    volatile uint32_t input_val;  // (GPIO смещение 0x00) Уровень на выводе
    volatile uint32_t input_en;   // (GPIO смещение 0x04) Вывод как вход*
    volatile uint32_t output_en;  // (GPIO смещение 0x08) Вывод как выход*
    volatile uint32_t output_val; // (GPIO смещение 0x0C) Выходное значение
    volatile uint32_t pue;        // (GPIO смещение 0x10) Включение подтягивающих резисторов*
    volatile uint32_t ds;         // (GPIO смещение 0x14) Мощность выхода
    volatile uint32_t rise_ie;    // (GPIO смещение 0x18) Прерывание по переднему фронту сигнала разрешено
    volatile uint32_t rise_ip;    // (GPIO смещение 0x1C) Ожидание прерывания по переднему фронту сигнала
    volatile uint32_t fall_ie;    // (GPIO смещение 0x20) Прерывание по заднему фронту сигнала разрешено
    volatile uint32_t fall_ip;    // (GPIO смещение 0x24) Ожидание прерывания по заднему фронту сигнала
    volatile uint32_t high_ie;    // (GPIO смещение 0x28) Прерывание по высокому уровню сигнала разрешено
    volatile uint32_t high_ip;    // (GPIO смещение 0x2C) Ожидание прерывания по высокому уровню сигнала
    volatile uint32_t low_ie;     // (GPIO смещение 0x30) Прерывание по низкому уровню сигнала разрешено
    volatile uint32_t low_ip;     // (GPIO смещение 0x34) Ожидание прерывания по низкому уровню сигнала
    volatile uint32_t iof_en;     // (GPIO смещение 0x38) Аппаратные функции разрешены
    volatile uint32_t iof_sel;    // (GPIO смещение 0x3C) Выбор аппаратных функций
    volatile uint32_t out_xor;    // (GPIO смещение 0x40) Операция XOR с выходом (инверсия)
    // Регистры, обозначенные звездочкой *, асинхронно сбрасываются в 0 при включении
} GPIO;

// Определяем базовый адрес регистров GPIO (табл. 9.1) и указатель на эту структуру
// Запись 0x…U в 0x10012000U означает шестнадцатеричное число без знака
#define GPIO0_BASE (0x10012000U)
#define GPIO0 ((GPIO*) GPIO0_BASE)

// Для доступа к элементу структуры применяется оператор ->.

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
```

Пример кода 9.2 (окончание)

```
            GPIO0->input_en  |=  (1 << gpio_pin); // Настраиваем вывод как вход
            GPIO0->output_en &= ~(1 << gpio_pin); // Сбрасываем бит output_en
            GPIO0->iof_en    &= ~(1 << gpio_pin); // Выключаем IOF
            break;
        case OUTPUT:
            GPIO0->output_en |=  (1 << gpio_pin); // Настраиваем вывод как выход
            GPIO0->input_en  &= ~(1 << gpio_pin); // Сбрасываем бит input_en
            GPIO0->iof_en    &= ~(1 << gpio_pin); // Выключаем IOF
            break;
    }
}

void digitalWrite(int gpio_pin, int val) {
    if (val) GPIO0->output_val |=  (1 << gpio_pin);
    else     GPIO0->output_val &= ~(1 << gpio_pin);
}

int digitalRead(int gpio_pin) {
    return (GPIO0->input_val >> gpio_pin) & 0x1;
}

// Приведенный ниже код читает состояние кнопок и включает или выключает светодиоды.
// Он настраивает выводы 2-4 как входы для чтения состояния кнопок и выводы 7-9 как выходы
// для управления светодиодами. Затем записывает прочитанное значение с вывода кнопки в выход,
// подключенный к соответствующему светодиоду.

#include "EasyREDVIO.h"
int main(void) {
  // Настроить GPIO 4:2 как входы
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);
  // Настроить GPIO 10:8 как выходы
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  while (1) { // Читаем каждую кнопку и передаем значение на вывод светодиода
    digitalWrite(8, digitalRead(2));
    digitalWrite(9, digitalRead(3));
    digitalWrite(10, digitalRead(4));
  }
}
```

9.3.5. Последовательный ввод/вывод

Микроконтроллер может отправить несколько битов на внешнее устройство двумя способами – по нескольким проводам одновременно или посылая один бит за другим по одному проводу. Первый способ называется *параллельным вводом-выводом*, а второй – *последовательным вводом-выводом*. Последовательный ввод/вывод более популярен, особенно когда количество выводов ограничено, потому что он использует небольшое число проводов, но при этом его скорости достаточно для множества задач. На самом деле он настолько популярен, что для последовательного ввода/вывода было разработано множество стандартов,

а микроконтроллеры содержат специальные аппаратные модули для простой передачи данных в соответствии с этими стандартами. Этот раздел описывает протоколы последовательного периферийного интерфейса (serial peripheral interface, SPI) и универсального асинхронного приемопередатчика (universal asynchronous receiver/transmitter, UART).

Другими распространенными последовательными стандартами являются двухпроводная двунаправленная шина (inter-integrated circuit, I^2C), универсальная последовательная шина (universal serial bus, USB) и Ethernet. I^2C – двухпроводной интерфейс с тактирующим сигналом и двунаправленным портом данных; он используется примерно тем же образом, что и SPI. USB и Ethernet – более сложные высокопроизводительные стандарты. FE310 поддерживает SPI, UART и I^2C при помощи встроенных периферийных модулей.

Последовательный периферийный интерфейс (SPI)

SPI (произносится как «эс-пи-ай») – простой синхронный последовательный протокол, легкий в использовании и относительно быстрый. Физический интерфейс состоит из трех линий: Serial Clock (SCK), Serial Data Out (SDO) и Serial Data In (SDI). SPI соединяет контроллер с периферийным устройством, как показано на **рис. 9.6 (a)**. Контроллер выдает тактовые импульсы. Он инициирует обмен данными, посылая тактовые импульсы по линии SCK. Контроллер передает данные со своего вывода SDO на вывод SDI периферийного устройства по одному биту за такт, начиная со старшего бита. Периферийное устройство может одновременно отправлять данные со своего вывода SDO на вывод SDI контроллера. На **рис. 9.6 (b)** показана диаграмма сигналов SPI для передачи 8 бит данных. Смена битов происходит по заднему фронту импульса SCK, и новый бит готов к считыванию по переднему фронту следующего импульса. В интерфейсе SPI может также применяться сигнал разрешения с активным низким уровнем, чтобы предупредить приемник о поступлении данных.

Термины «ведущий/ведомый» (master/slave) раньше были общепринятыми (вместо «контроллер/периферийное устройство»), но теперь они устарели. Вместо обозначения MOSI (master-out slave-in, выход ведущего – вход ведомого) теперь используется термин «*последовательный выход данных*» (serial data out, SDO) или «*выход контроллера – вход периферии*» (controller out – peripheral in, COPI). Вместо MISO (master-in slave-out) теперь используется термин «*последовательный вход данных*» (serial data in, SDI) или «*вход контроллера – выход периферии*» (controller-in peripheral-out, CIPO).

Термин «порт» относится к контакту или группе связанных контактов.

FE310 имеет три порта контроллера SPI, но пользователю доступны только два (SPI1 и SPI2). Порт SPI0 используется для связи с внешней флеш-памятью для хранения программ и данных. В этом разделе описывается порт контроллера SPI1, доступ к которому осуществляется через контакты GPIO 5:2. Порт контроллера SPI2 идентичен, за исключением того, что он подключен к другим выводам GPIO, а его регистры управления расположены по другим адресам памяти. Чтобы использовать выводы как порт SPI, а не GPIO, их биты в регистре `iof_sel` должны быть сброшены (выбор функции SPI1), а их биты `iof_en` должны быть установ-

лены, дабы обеспечить контроллеру SPI доступ к выводам. После того как FE310 записал данные в регистр SPI `txdata`, они последовательно передаются на ведомое устройство. Одновременно с этим собираются данные, полученные от ведомого устройства, и FE310 может прочитать их из регистра `rxdata` после завершения передачи.

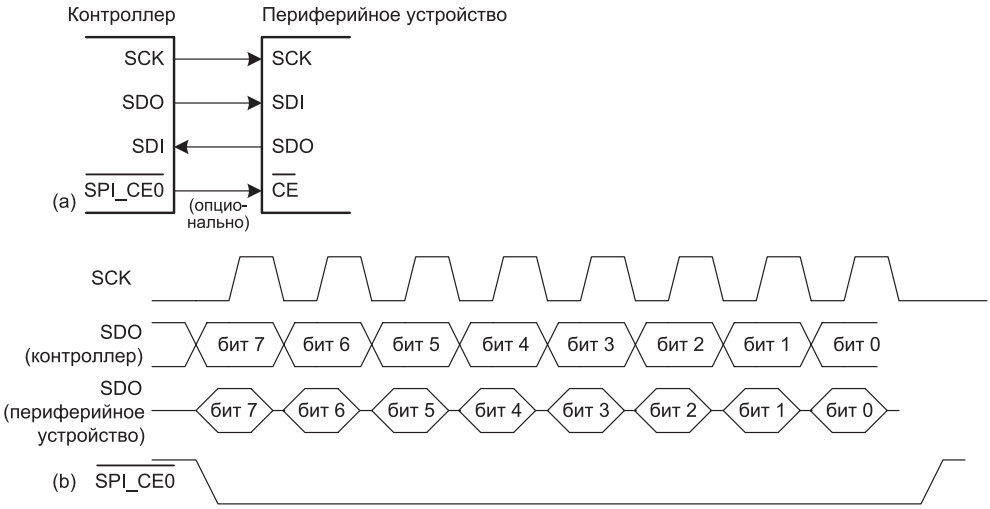


Рис. 9.6 Конфигурация SPI: (a) схема подключения контроллера SPI к периферии, (b) диаграмма сигналов SPI

SPI всегда отправляет данные в обоих направлениях при каждой передаче. Если системе требуется только однонаправленная связь, она может игнорировать нежелательные данные. Например, если контроллеру нужно лишь отправить данные на ведомое устройство, то байт, полученный от ведомого устройства, можно игнорировать. Если контроллеру нужно только получать данные от ведомого устройства, он все равно должен инициировать обмен по SPI, отправляя произвольный байт, который ведомое устройство будет игнорировать. Затем он может прочитать данные, полученные от ведомого устройства. Тактовые импульсы SPI (SCK) вырабатываются только тогда, когда контроллер передает данные.

Порты SPI микроконтроллера обычно поддерживают множество вариантов конфигурации, чтобы соответствовать требованиям ведомых устройств. При разработке интерфейса для связи с конкретным ведомым устройством контроллер должен быть правильно настроен, чтобы гарантировать безошибочную интерпретацию передаваемых данных.

Два основных параметра конфигурации – это *полярность тактового сигнала* (CPOL) и *фаза тактового сигнала* (CPHA). CPOL определяет логический уровень на выходе тактовых импульсов в режиме ожидания, а CPHA задает фронты тактовой частоты, по которым происходят выборка и изменение данных на линиях SDO и SDI. Если CPOL = 1, то в режиме ожидания SCK остается на высоком уровне (1); если CPOL = 0, SCK остается низким (0). Если CPHA = 0, данные выбираются по переднему фронту и изменяются по заднему фронту тактового импульса SCK; если CPHA = 1, данные выбираются по заднему фронту и изменяются по переднему фронту

SCK. Фронт тактового импульса, по которому изменяются данные, также называется *фронтом сдвига*, потому что лежащее в основе модуля SPI оборудование обычно представляет собой сдвиговый регистр. На **рис. 9.7** показаны четыре возможные комбинации CPHA и CPOL. На **рис. 9.6** дано сочетание CPOL = 0 и CPHA = 0.

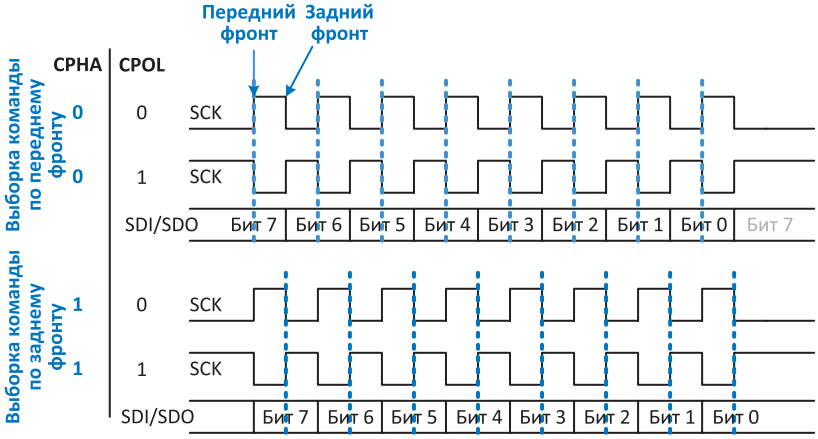


Рис. 9.7 Временная диаграмма и биты конфигурации различных режимов SPI

В **табл. 9.4** показаны регистры управления, связанные с SPI1, а в **табл. 9.5** представлены поля ключевых регистров. Регистр `sckdiv` (**табл. 9.4**) определяет тактовую частоту SPI, указывая делитель (`div`) для исходной тактовой частоты – на плате RED-V частота тактового генератора по умолчанию составляет 16 МГц. Частота тактовых импульсов SPI вычисляется по формуле $f_{sck} = \frac{f_{in}}{2(div+1)}$. Например, если `div` = 15, то $f_{sck} = \frac{16\ \text{МГц}}{2(15+1)} = 500$ кГц. Если частота слишком высокая (превышает 1 МГц на макетной плате или десятки МГц на печатной плате), соединение SPI может работать со сбоями из-за отражений, взаимных помех или других недостатков сигнала.

Содержимое регистра `sckmode` определяет фазу и полярность тактовых импульсов. Задействованы только два младших разряда регистра: бит 0 – это CPHA, а бит 1 – это CPOL.

В регистр `txdata` записывают байт, который нужно передать по каналу SPI, а из регистра `rxdata` считывают принятый байт. Передается только младший байт (LSB), находящийся в `txdata`. В модуле SPI микроконтроллера FE310 как регистр передачи, так и регистр приема данных работают по принципу буфера FIFO (first-in-first-out, первый вошел – первый вышел) на 8 элементов. Это означает, что при записи данных в регистр `txdata` они помещаются в буфер FIFO, и аппаратное

обеспечение периферийного модуля SPI само заботится об их отправке. Самый старший бит (msb) регистра `txdata` – это бит флага `full`, который равен единице, когда буфер FIFO заполнен и больше не может принимать данные.

Таблица 9.4 Карта памяти регистров SPI

Смещение	Имя регистра	Описание
0X00	sckdiv	Делитель тактовой частоты
0X04	sckmode	Режим тактовой частоты
0X08	Зарезервирован	
0X0C	Зарезервирован	
0X10	csid	Номер микросхемы (ID)
0X14	csdef	Выбор микросхемы по умолчанию
0X18	csmode	Режим выбора микросхемы
0X1C	Зарезервирован	
0X20	Зарезервирован	
0X24	Зарезервирован	
0X28	delay0	Управление задержкой, регистр 0
0X2C	delay1	Управление задержкой, регистр 1
0X30	Зарезервирован	
0X34	Зарезервирован	
0X38	Зарезервирован	
0X3C	Зарезервирован	
0X40	fmt	Формат кадра
0X44	Зарезервирован	
0X48	txdata	Данные передачи (Tx FIFO)
0X4C	rxdata	Данные приема (Rx FIFO)
0X50	txmark	Tx FIFO watermark (метка заполненности FIFO)
0X54	rxmark	Rx FIFO watermark
0X58	Зарезервирован	
0X5C	Зарезервирован	
0X60	fctrl	Управление SPI флеш-интерфейсом*
0X64	ffmt	Формат SPI флеш-команды*
0X68	Зарезервирован	
0X6C	Зарезервирован	
0X70	ie	Разрешение прерывания SPI
0X74	ip	Запрос прерывания SPI

Перепечатано из табл. 65 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Необходимо соблюдать осторожность при чтении данных из регистра `rxdata`. Контроллер SPI спроектирован таким образом, что принятые данные удаляются из приемного буфера FIFO при чтении регистра. Чтобы проверить, есть ли в регистре `rxdata` корректные данные, следует

однократно прочитать регистр, а затем проверить бит флага опустошения, дабы определить, действительны ли эти данные. Программисту следует позаботиться о том, чтобы не читать регистр более одного раза для каждого байта, так как это приведет к потере данных.

Таблица 9.5 Битовые поля регистров SPI

Регистр делителя тактовой частоты (sckdiv)				
Смещение адреса		0x0		
Биты	**Имя поля**	**Атрибут**	**Сброс**	**Описание**
[11:0]	div	RW	0x3	Делитель тактовой частоты шириной div_width бит
[31:12]	Зарезервировано			

Регистр режима тактовых импульсов (sckmode)				
Смещение адреса		0x4		
Биты	**Имя поля**	**Атрибут**	**Сброс**	**Описание**
0	pha	RW	0x0	Фаза тактовых импульсов
1	pol	RW	0x0	Полярность тактовых импульсов
[31:2]	Зарезервировано			

Регистр передаваемых данных (txdata)				
Смещение адреса		0X48		
Биты	**Имя поля**	**Атрибут**	**Сброс**	**Описание**
[7:0]	data	RW	0X0	Передаваемые данные
[30:8]	Зарезервировано			
31	full	RO	X	Флаг заполнения FIFO

Регистр принимаемых данных (rxdata)				
Смещение адреса		0X4C		
Биты	**Имя поля**	**Атрибут**	**Сброс**	**Описание**
[7:0]	data	RO	X	Принимаемые данные
[30:8]	Зарезервировано			
31	empty	RW	X	Флаг освобождения FIFO

Перепечатано из табл. 66, 67, 80 и 81 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Регистры конфигурации имеют много неиспользуемых, или «зарезервированных», битов. Эти биты могут быть использованы в будущей версии микросхемы, поэтому их не следует записывать, чтобы не вызвать непредвиденные последствия в будущем.

Дополнительные регистры csid, csdef и csmode определяют параметры, связанные с управлением и конфигурацией линии выбора микросхемы. В качестве альтернативы вывод выбора микросхемы (CS) можно сконфигурировать как выход GPIO и управлять им программно при помощи digitalWrite.

Некоторые регистры SPI упаковывают несколько небольших полей данных в одно 32-битное слово. В программе на языке C мы можем объявить количество битов каждого поля с помощью двоеточия и числа как часть структуры битового поля. В **примере 9.3** показано, как использовать битовые поля и структуры для определения этих регистров.

Пример 9.3 ОТПРАВКА И ПРИЕМ БАЙТОВ ПО SPI

Разработайте систему для связи между контроллером FE310 и ведомым устройством FPGA через SPI. Нарисуйте схему интерфейса. Разработайте код на языке C для FE310, чтобы отправить символ «A» и получить один символ обратно. Напишите HDL-код для ведомого устройства SPI на FPGA. Как можно упростить ведомое устройство, если ему нужно только получать данные?

Решение На **рис. 9.8** показана схема канала обмена данными между контроллером FE310 и ведомым устройством FPGA с использованием SPI1. Номера выводов получены из технической документации компонентов (например, **табл. 9.3** для FE310). Обратите внимание, что на схеме показаны как номера контактов, так и названия сигналов, чтобы описать как физическую, так и логическую часть соединения. Когда активен модуль SPI, эти выводы нельзя использовать для GPIO.

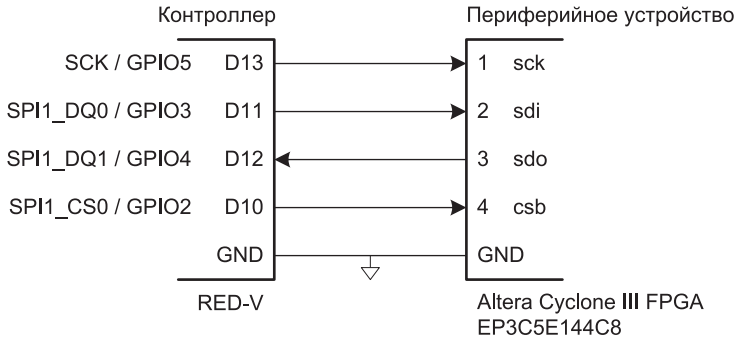


Рис. 9.8 Схема подключения RED-V к FPGA Altera Cyclone

Представленный ниже код из файла EasyREDVIO.h применяется для инициализации канала SPI, а также для отправки и получения символа. В файле сначала объявляются битовые поля SPI и карта памяти. Функция `pinMode` расширена и теперь поддерживает как интерфейс ввода/вывода, так и обмен данными. Функция `spiSendReceive` выполняет отправку и получение одного байта транзакции SPI. Сначала она проверяет, не заполнен ли передающий буфер FIFO и может ли он принять другой символ. Если да, то функция записывает символ в передающий буфер FIFO, который при этом сдвигается. После передачи считывается регистр `rxdata`. Здесь следует проявлять осторожность, потому что бит флага освобождения регистра `rxdata` обновляется при каждом чтении регистра. Поэтому необходимо прочитать весь 32-битный регистр `rxdata`. Затем, после проверки того, что флаг (регистр пустой) не установлен (т. е. данные корректные), функция возвращает полученный байт.

Пример кода 9.3 ВЫЗОВ ФУНКЦИЙ ДЛЯ РАБОТЫ SPI

```
////////////////////////////////////////////////////////////////////////////
// Регистры SPI
////////////////////////////////////////////////////////////////////////////
typedef struct {
    volatile uint32_t       div     : 12;   // Делитель тактовой частоты
    volatile uint32_t : 20;
} sckdiv_bits;

typedef struct {
    volatile uint32_t       pha     : 1;    // Фаза тактовых импульсов
    volatile uint32_t       pol     : 1;    // Полярность тактовых импульсов
    volatile uint32_t               : 30;
} sckmode_bits;

...

typedef struct {
    volatile uint32_t       data    : 8;    // Передаваемые данные
    volatile uint32_t               : 23;
    volatile uint32_t       full    : 1;    // Флаг заполнения FIFO
} txdata_bits;

typedef struct {
    volatile uint32_t       data    : 8;    // Принятые данные
    volatile uint32_t               : 23;
    volatile uint32_t       empty   : 1;    // Флаг освобождения FIFO
} rxdata_bits;

// Режимы выводов
#define INPUT 0
#define OUTPUT 1
#define GPIO_IOF0 2
#define GPIO_IOF1 3

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
             GPIO0->input_en |= (1 << gpio_pin);  // Назначить вывод как вход
             GPIO0->iof_en &= ~(1 << gpio_pin);   // Отключить IOF
             break;
        case OUTPUT:
             GPIO0->output_en |= (1 << gpio_pin); // Назначить вывод как выход
             GPIO0->iof_en &= ~(1 << gpio_pin);   // Отключить IOF
             break;
        case GPIO_IOF0:
             GPIO0->iof_en |= (1 << gpio_pin);    // Включить IOF
             GPIO0->iof_sel &= ~(1 << gpio_pin);  // Функция IO 0
             break;
        case GPIO_IOF1:
             GPIO0->iof_en |= (1 << gpio_pin);    // Включить IOF
             GPIO0->iof_sel |= (1 << gpio_pin);   // Функция IO 1
             break;
    }
}

void spiInit(uint32_t clkdivide, uint32_t cpol, uint32_t cpha) {

    // Инициализировать выводы SPI (GPIO 2-5) как выводы аппаратного модуля
    pinMode(3, GPIO_IOF0); // SDO
    pinMode(4, GPIO_IOF0); // SDI
    pinMode(5, GPIO_IOF0); // SCK
```

Пример кода 9.3 (окончание)

```
    digitalWrite(2, 1); // Проверяем, что на CS0 не низкий уровень сигнала
    pinMode(2, OUTPUT); // Состояние CS0 задается принудительно

    SPI1->sckdiv.div = clkdivide; // Настройка делителя частоты

    SPI1->sckmode.pol = cpol;     // Настройка полярности
    SPI1->sckmode.pha = cpha;     // Настройка фазы
}

/* Передача символа (1 байт) через SPI и чтение полученного символа.
* send: символ для передачи через SPI
* return value: символ, принятый через SPI */
uint8_t spiSendReceive(uint8_t send) {
    while(SPI1->txdata.full); // Ждем, пока передающий FIFO не будет готов принять
                              // новый байт
    SPI1->txdata.data = send; // Передаем символ через SPI

    rxdata_bits rxdata;
    while (1) {
        rxdata = SPI1->rxdata; // ОДНОКРАТНО читаем регистр rxdata
        if (!rxdata.empty) {   // Если флаг опустошения не был установлен,
                               // возвращаем прочитанный символ
            return (uint8_t)rxdata.data;
        }
    }
}
```

Код на языке C в **примере кода 9.4** инициализирует SPI, а затем отправляет и получает символ. Исходя из формулы $f_{sck} = \frac{f_{in}}{2(div+1)}$, где f_{in} – это тактовая частота ядра `coreclk` = 16 МГц, он устанавливает тактовую частоту SPI равной 500 кГц.

Пример кода 9.4 ВЫЗОВ ФУНКЦИЙ ДЛЯ РАБОТЫ SPI

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t volatile received;

  // Инициализация SPI
  // Делитель частоты для div = 15, CPOL = 0, CPHA = 0
  spiInit(15, 0, 0);

  digitalWrite(2, 0);             // Выбор ведомого устройства (CS = 0), если необходимо
  received = spiSendReceive('A'); // Передача символа A и прием байта
  digitalWrite(2, 1);             // Отключение сигнала выбора ведомого устройства
}
```

Если ведомое устройство должно только получать данные от контроллера, то приемник представляет собой простой сдвиговый регистр, как показано в **HDL-примере 9.1**. На каждом переднем фронте `sck` новый бит `sdi` сдвигается в регистр сдвига, начиная с самого старшего бита данных. После восьми тактов весь байт был считан в регистр сдвига.

HDL-пример 9.1 HDL-КОД ПРИЕМНИКА SPI ВНЕШНЕГО УСТРОЙСТВА

```
module spi_peripheral_receive_only(input logic sck,       // Входной тактовый сигнал от контроллера
                                   input logic sdi,       // Входной сигнал данных от контроллера
                                   output logic [7:0] q); // Принятые данные
  always_ff @(posedge sck)
    q < = {q[6:0], sdi}; // сдвиговый регистр
endmodule
```

В **HDL-примере 9.2** представлен код SystemVerilog для ведомого устройства SPI, которое может как отправлять, так и получать данные (как, например, приемопередатчик SPI), а на **рис. 9.9** показана его блок-схема и временная диаграмма для CPHA = CPOL = 0. Основным компонентом по-прежнему является сдвиговый регистр, показанный справа на **рис. 9.9**. Этот регистр получает байт для отправки по параллельной шине (`d[7:0]`), а затем сдвигает эти данные в `sdo`, одновременно сдвигая данные, поступающие от контроллера (`t[7:0]`) на `sdi`. Счетчик `cnt` отслеживает, сколько битов было отправлено/получено. Когда `sck` простаивает, `cnt` = 0 и значение старшего бита `d(d[7])` остается на линии `sdo`. Особенность такой реализации заключается в том, что `sdo` может изменяться только по заднему фронту тактового импульса, поэтому выход `sdo` (который является самым старшим битом сдвигового регистра `q[7]`) задерживается на половину такта с помощью регистра, управляемого по заднему фронту `qdelayed`, который расположен в нижнем левом углу **рис. 9.9**.

HDL-пример 9.2 HDL-КОД ПРИЕМОПЕРЕДАТЧИКА SPI ВЕДОМОГО УСТРОЙСТВА

```
module spi_peripheral(input logic sck,       // От контроллера
                      input logic sdi,       // От контроллера
                      output logic sdo,      // К контроллеру
                      input logic reset,     // Системный сброс
                      input logic [7:0] d,   // Передаваемые данные
                      output logic [7:0] q); // Принимаемые данные
  logic [2:0] cnt;
  logic  qdelayed;

  // 3-битный счетчик отслеживает, когда будет передан полный байт данных
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else cnt = cnt + 3'b1;

  // Сдвиговый регистр с возможностью установки состояния
  // Загружает d в начале работы схемы. На каждом шаге сдвигает бит sdi в позицию младшего бита
  always_ff @(posedge sck)
    q < = (cnt = = 0) ? {d[6:0], sdi} : {q[6:0], sdi};

  // Выравниваем sdo по заднему фронту sck
  // Загружаем d в начале передачи
  always_ff @(negedge sck)
    qdelayed = q[7];

  assign sdo = (cnt = = 0) ? d[7] : qdelayed;
endmodule
```

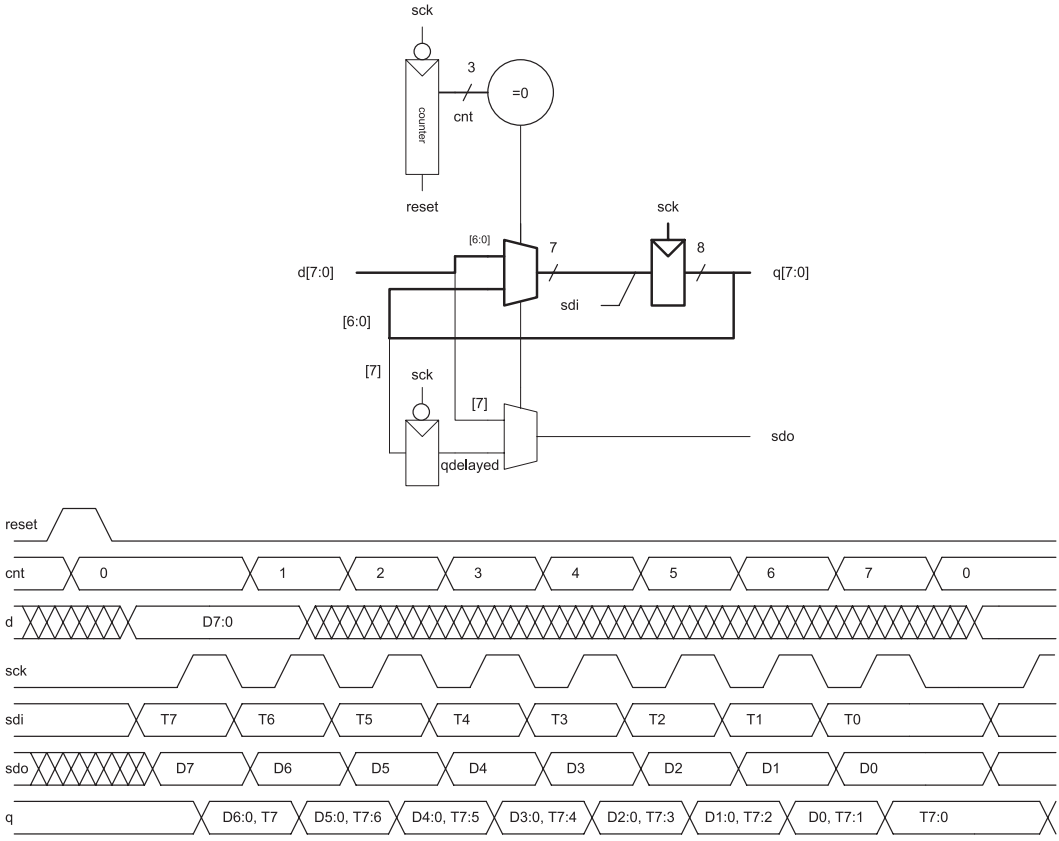


Рис. 9.9 **Блок-схема и временная диаграмма приемопередатчика ведомого устройства SPI**

Универсальный асинхронный приемопередатчик (UART)

UART (произносится как «у-арт») – это периферийное устройство последовательного ввода/вывода для обмена данными между устройствами без использования тактового сигнала. Вместо этого системы должны заранее согласовать скорость передачи данных, и каждая из них должна локально генерировать свой собственный тактовый сигнал. Следовательно, такая передача является *асинхронной*, потому что тактовые сигналы не синхронизированы. Хотя системные тактовые сигналы могут иметь небольшую погрешность частоты и неизвестное соотношение фаз, UART обеспечивает надежную асинхронную связь. UART используется в таких протоколах, как RS-232 и RS-485. Например, последовательные порты старых компьютеров используют стандарт RS-232C, введенный в 1969 году Ассоциацией электронной промышленности. Стандарт пер-

воначально предполагалось использовать для подключения пользовательского *устройства обработки данных* (data terminal equipment, DTE), такого как мейнфреймы, к *устройству передачи данных* (data communication equipment, DCE), например модемам. Хотя UART работает относительно медленно по сравнению с SPI, этот стандарт применялся так долго, что он остается важным и сегодня.

На **рис. 9.10 (a)** показан асинхронный последовательный канал. DTE передает данные в DCE по линии TX и получает данные обратно по линии RX. На **рис. 9.10 (b)** показана одна из этих линий при передаче данных со скоростью 9600 *бод*. В состоянии ожидания на линии присутствует постоянный уровень логической единицы. Каждый символ состоит из стартового бита (0), 7 или 8 бит данных, опционального бита четности и одного или нескольких стоповых битов (1). Чаще всего передают стартовые и стоповые биты и восемь бит данных. UART обнаруживает переход от состояния ожидания к началу передачи, чтобы своевременно начать процесс передачи данных. Хотя семи бит данных достаточно для отправки символа ASCII, обычно используются восемь бит, поскольку они позволяют передавать произвольный байт данных.

Скорость передачи сигнала, измеряемая в бодах, — это количество передаваемых символов в секунду, тогда как *скорость передачи данных* измеряют в битах в секунду. В простой системе, такой как SPI, где каждый символ одновременно является битом данных, скорость передачи символов равна скорости передачи данных. В UART и некоторых других стандартах передачи сигналов в дополнение к данным передают служебные биты. Например, UART, который добавляет стартовые и стоповые биты для каждых 8 бит данных (т. е. передает 10 символов на 8 бит данных) и работает со скоростью 9600 бод, имеет скорость передачи (9600 символов/с)×(8 бит/10 символов)= 7680 бит/с = 960 знаков/с.

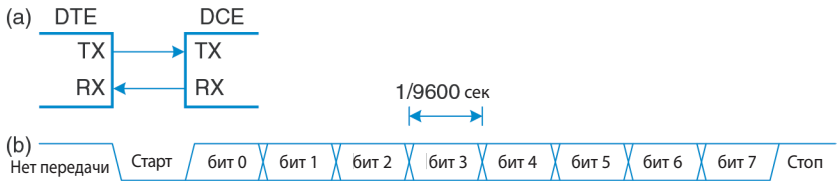


Рис. 9.10 Асинхронный последовательный канал

Дополнительный бит четности позволяет системе определить, не повреждены ли данные во время передачи. Систему можно настроить на *контроль четности* или *нечетности*. В первом случае бит четности выбирается таким образом, чтобы общий набор данных, включая бит четности, содержал четное количество единиц. Другими словами, бит четности — это результат операции XOR над битами данных. Приемник проверяет, было ли получено четное количество единиц, и в противном случае сигнализирует об ошибке. Контроль нечетности работает противоположным образом.

На практике обычно используют 1 стартовый бит, 8 бит данных без контроля четности и 1 стоповый бит, что в сумме дает 10 символов для передачи 8-битного знака информации. По этой причине скорости пере-

В 1950–1970-х годах первые хакеры, называвшие себя телефонными фрикерами, научились имитировать тональные управляющие сигналы телефонных компаний свистом соответствующего тона. Тон частотой 2600 Гц, производимый игрушечным свистком из коробки с кукурузными хлопьями «Капитан Кранч» (рис. 9.11), можно было использовать для осуществления бесплатной междугородной и международной связи.

Рис. 9.11 Боцманский свисток капитана Кранча (перепечатано с разрешения Эврима Сена)

дачи указывают в бодах, а не в битах в секунду. Например, 9600 бод означает 9600 символов в секунду, или 960 знаков в секунду. И передатчик, и приемник должны быть настроены на одинаковую скорость передачи и количество битов данных, четность и стоповые биты, иначе данные будут искажены. Это лишняя сложность, особенно для пользователей без хорошей технической подготовки, что является одной из причин, по которой USB заменил UART в персональных компьютерах.

Типичные скорости передачи данных по UART включают 300, 1200, 2400, 9600, 14 400, 19 200, 38 400, 57 600 и 115 200 бод. Малые значения скорости использовались в 1970-х и 1980-х годах для модемов, которые посылали данные по телефонным линиям как последовательности тонов. В современных системах 9600 и 115 200 являются двумя наиболее распространенными скоростями передачи; 9600 встречается там, где скорость не имеет значения, а 115 200 остается самой высокой стандартной скоростью, хотя она меньше по сравнению с другими современными стандартами последовательного ввода/вывода.

Стандарт RS-232 определяет несколько дополнительных сигналов. Сигналы запроса на передачу (request to send, RTS) и готовности к передаче (clear to send, CTS) могут использоваться для аппаратного подтверждения связи (handshaking). Они могут работать в одном из двух режимов. В *режиме управления потоком* DTE сбрасывает RTS в 0, когда он готов к приему данных от DCE. Точно так же DCE устанавливает CTS в 0, когда он готов к приему данных от DTE. Иногда в технической документации используют символ надчеркивания (верхней черты), чтобы указать, что разрешающий сигнал – низкого уровня. В старом *симплексном* (одностороннем) режиме передачи DTE обнуляет RTS, когда он готов к передаче. DCE отвечает обнулением CTS, когда он готов принимать передачу.

Некоторые системы, особенно подключенные по телефонной линии, также использовали сигналы готовности терминала данных (DTR), обнаружения носителя данных (DCD), готовности набора данных (DSR) и индикатора звонка (RI), чтобы указать, когда оборудование подключено к линии. Эти сигналы все еще присутствуют в некоторых разъемах.

Процесс подтверждения связи (*англ.* handshaking – рукопожатие) относится к «переговорам» между двумя системами. Как правило, одна система сигнализирует, что она готова к отправке или получению данных, а другая система подтверждает этот запрос.

Оригинальный стандарт RS-232 описывает массивный 25-контактный разъем DB-25, но в ПК он превратился в 9-контактный штекерный разъем DE-9 с разводкой выводов, показанной на **рис. 9.12 (a)**. Сигнальные линии кабеля обычно подключаются напрямую, как показано на **рис. 9.12 (b)**. Но при непосредственном соединении

двух DTE может потребоваться *нуль-модемный кабель*, показанный на **рис. 9.12 (c)**, в котором контакты RX и TX соединены крест-накрест, как и контакты линий подтверждения связи. Вдобавок некоторые разъемы бывают штыревыми, а некоторые гнездовыми. Таким образом, для соединения двух систем через RS-232 вам может потребоваться большой пучок кабелей и определенное везение, что служит еще одним объяснением перехода на USB. К счастью, встраиваемые системы обычно используют упрощенную 3- или 5-проводную схему, состоящую из линий GND, TX, RX и, иногда, RTS и CTS.

В оригинальном стандарте RS-232 логическому нулю соответствует напряжение в диапазоне от 3 до 15 В, а логической единице – напряжение от −3 до −15 В; это называется биполярной передачей сигналов. Приемопередатчик преобразует цифровые логические уровни UART в положительные и отрицательные напряжения в соответствии с RS-232, а также обеспечивает защиту последовательного порта от повреждения электростатическим разрядом в момент подключения кабеля. Микросхема MAX3232E – популярный приемопередатчик, совместимый с цифровой логикой как 3,3, так и 5 В. Он содержит схему накачки заряда, которая вместе с внешними конденсаторами генерирует выходное напряжение ± 5 В от одного низковольтного источника питания. Некоторые последовательные устройства, предназначенные для встраиваемых систем, не имеют приемопередатчика и просто работают со стандартными цифровыми логическими уровнями системы; при разработке системы сверяйтесь с техническим описанием!

Микроконтроллер FE310 имеет два встроенных модуля UART с именами UART0 и UART1. Модуль UART0 можно настроить для работы через выводы 16 и 17; UART1 использует выводы 18 и 23. Чтобы использовать эти выводы в качестве UART, а не GPIO, соответствующие биты регистра `iof_sel` должны быть сброшены (чтобы выбрать `IOF0`), а биты регистра `iof_en` – установлены, чтобы разрешить периферийному модулю управлять выводами. Как и в случае с SPI, процессор FE310 должен сначала настроить порт. В отличие от SPI, чтение и запись могут происходить независимо, поскольку любая сторона системы может передавать данные без приема, и наоборот. Регистры UART0 показаны в **табл. 9.6**.

Чтобы настроить UART, сначала установите скорость передачи. В качестве источника тактовой частоты UART использует тактовые импульсы `tlclk` внутренней шины `TileLink`. Для FE310-G002 частота шины по умолчанию настроена так, чтобы она совпадала с частотой тактирования процессора `coreclk`, равной 16 МГц. Эту тактовую частоту необходимо разделить, чтобы получить нужную скорость передачи данных. Конечная скорость передачи данных определяется **уравнением 9.1**:

$$f_{baud} = \frac{f_{in}}{div + 1}. \tag{9.1}$$

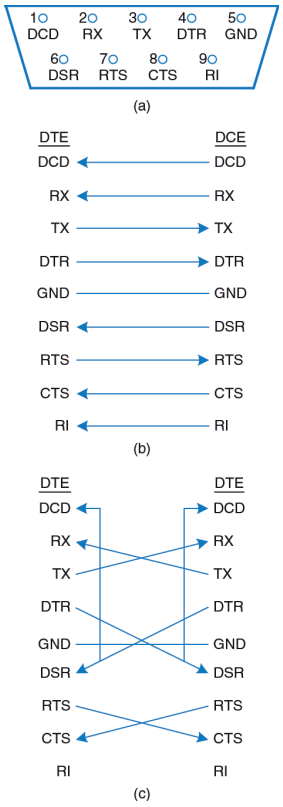


Рис. 9.12 Кабель DE-9: (a) расположение выводов, (b) стандартная разводка и (c) нуль-модемная разводка

Таблица 9.6. Регистры UART0, отображаемые в память

	...
0x10013018	div
	...
0x1001300C	rxctrl
0x10013008	txctrl
0x10013004	rxdata
0x10013000	txdata
	...

Встроенный модуль UART микроконтроллера FE310 поддерживает только конфигурации протокола 8-N-1 и 8-N-2. Обе конфигурации поддерживают 8 бит данных и не поддерживают бит четности, а для пакетов можно настроить один стоповый бит (8-N-1) или два стоповых бита (8-N-2). Конфигурация стопового бита устанавливается в регистре `txctrl` с помощью поля `nstop`. По умолчанию `nstop` = 0, что указывает встроенному модулю UART использовать один стоповый бит.

Данные передаются и принимаются с использованием регистров `txdata` и `rxdata` соответственно. Регистры передачи и приема функционально представляют собой буферы FIFO на 8 элементов. Перед передачей данных убедитесь, что бит заполнения регистра `txdata` равен нулю, что свидетельствует о наличии в буфере FIFO места для записи новых данных. Затем запишите байт в поле данных регистра `txdata`. Выполняя прием данных, прочтите регистр `rxdata` и сначала убедитесь, что бит заполненности буфера равен нулю, что означает наличие действительных данных в буфере приемника.

Пример 9.4 ПОСЛЕДОВАТЕЛЬНЫЙ ОБМЕН ДАННЫМИ С ПК

Разработайте схему и программу на языке C для FE310, которая будет осуществлять обмен данными с персональным компьютером через последовательный порт со скоростью передачи 115 200 бод в формате: 8 бит данных, 1 стоп-бит и без контроля четности. На ПК должна быть запущена консольная программа, например PuTTY[1], для чтения и записи через последовательный порт. Программа должна попросить пользователя ввести произвольную строку, а затем вывести эту строку в консоль.

Решение На **рис. 9.13 (a)** показана обобщенная схема последовательного соединения, иллюстрирующая решение задачи преобразования уровня и разводки кабеля. Поскольку лишь на немногих современных компьютерах остались физические последовательные порты, мы воспользуемся последовательным адаптером USB – RS-232 DB9 с сайта plugable.com, показанным на **рис. 9.14**. Адаптер подключается через разъем DE-9 к выводам трансивера, преобразующего напряжения биполярных уровней RS-232 в логические уровни 3,3 В FE310. И микроконтроллер FE310, и ПК являются устройствами DTE, поэтому их контакты TX и RX должны быть соединены крест-накрест. Линии подтверждения связи RTS/CTS в FE310 не использу-

[1] Приложение PuTTY доступно для скачивания и свободного использования на сайте www.putty.org.

ются, а RTS и CTS на разъеме DE9 соединены перемычкой, так что ПК будет выдавать подтверждение связи самому себе.

На **рис. 9.13 (b)** показан более простой подход с использованием кабель-адаптера Adafruit 954 USB – TTL. Этот адаптер напрямую совместим с уровнями 3,3 В.

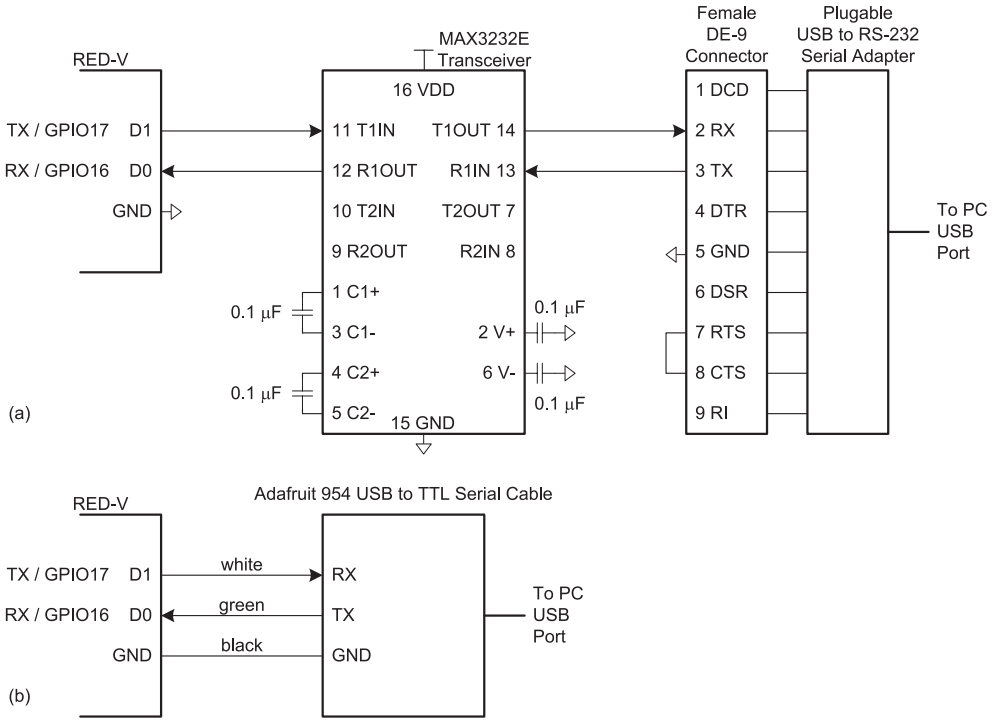


Рис. 9.13 Схема последовательного канала передачи данных: (a) последовательный обмен данными через RS-232, (b) последовательный обмен данными через адаптер USB – TTL

Рис. 9.14 Адаптер USB – RS-232 DB9

Чтобы настроить PuTTY для работы с последовательной линией связи, установите для параметра **Connection type** (Тип подключения) значение **Serial** (Последовательный), а параметр **Speed** (Скорость) равным 115 200. В поле **Serial line** (Последовательный канал) введите номер COM-порта, назначенного операционной системой для USB-адаптера. В Windows этот номер можно найти в диспетчере устройств; например, это может быть COM3. На вкладке **Connection → Serial** установите для контроля передачи значение NONE или RTS/CTS. На вкладке **Terminal** установите для параметра **Local echo** (Локальное эхо) значение **Force On**

(Принудительное включение), чтобы символы отображались в терминале по мере их ввода.

В **примере кода 9.5** приведен код драйвера устройства последовательного порта в EasyREDVIO.h. Клавиша **Enter** в программе терминала соответствует символу возврата каретки, представленному в языке C как `\r` с кодом ASCII 0x0D. Чтобы перейти к началу следующей строки при печати, отправьте символы `\n` и `\r` (новая строка и возврат каретки)[1]. Функция `uartInit` настраивает UART, как описано выше. Функции `getCharSerial` и `putCharSerial` считывают и отправляют символы в терминал, используя UART.

Пример кода 9.5 ДВУСТОРОННИЙ ОБМЕН СИМВОЛАМИ МЕЖДУ ТЕРМИНАЛОМ И ВНЕШНИМ УСТРОЙСТВОМ ПО UART

```
void uartInit(uint32_t baud) {
    uint32_t div = 16000000/baud-1;      // Тактовая частота 16 МГц
    pinMode(16, GPIO_IOF0);
    pinMode(17, GPIO_IOF0);

    UART0->div.div = div;                // Настройка делителя частоты
    UART0->txctrl.txen = 1;              // Включение приемопередатчика
    UART0->txctrl.nstop = 1;             // Один стоп-бит
    UART0->rxctrl.rxen = 1;              // Включение приемника
}

uint8_t getCharSerial(void) {
    uart_rxdata_bits rxdata;             // Создать временную переменную
                                         // для хранения регистра

    while(1) {
        rxdata = UART0->rxdata;          // ОДНОКРАТНО прочитать регистр
        if(!rxdata.empty) {
            return (uint8_t)rxdata.data; // Проверить, действительны ли данные
        }
    }
}

void putCharSerial(uint8_t c) {
    while(UART0->txdata.full);           // Ждать готовности передачи
    UART0->txdata.data = c;
}
```

Функция `main` в **примере кода 9.6** представляет собой пример, иллюстрирующий печать в консоль и чтение из консоли с помощью функций `putStrSerial` и `getStrSerial`.

Организация связи через последовательный порт из программы на языке C для ПК – это достаточно сложный процесс, потому что библиотеки драйверов последовательного порта не стандартизированы для разных операционных систем. Другие среды программирования, такие как Python, MATLAB или LabVIEW, позволяют организовать связь через последовательный порт с минимальными затратами усилий.

[1] PuTTY выполняет переход на новую строку корректно даже без символа `\r`.

Пример кода 9.6 ПРИЕМ И ПЕРЕДАЧА СТРОК ЧЕРЕЗ UART С ВВОДОМ И ВЫВОДОМ НА КОНСОЛЬ

```
#include "EasyREDVIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                            // Читаем строку до тех пор, пока
        str[i] = getCharSerial();   // не обнаружен символ возврата каретки
    } while ((str[i++] != '\r') && (i < MAX_STR_LEN)); // Ищем символ возврата каретки
    str[i-1] = 0;                   // Нулевое значение завершает строку
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {           // Перебор всех символов строки
        putCharSerial(str[i++]);    // Отправка каждого символа
    }
}

int main(void) {
    char str[MAX_STR_LEN];

    uartInit(115200);               // Инициализация UART с нужной скоростью

    while(1) {
        putStrSerial("Введите любую строку: \r\n");
        getStrSerial(str);
        putStrSerial("Вы ввели: ");
        putStrSerial(str);
        putStrSerial("\r\n");
    }
}
```

9.3.6. Таймеры

Встраиваемые системы обычно нуждаются в измерении времени. Например, микроволновой печи необходим один таймер для отслеживания времени суток и другой, чтобы определить, как долго готовить. Она может использовать третий таймер, чтобы генерировать импульсы для двигателя, вращающего блюдо, а четвертый – чтобы контролировать уровень мощности, активируя микроволны лишь на долю каждой секунды.

FE310 содержит системный таймер с 64-битным независимым счетчиком, значение которого асинхронно увеличивается в соответствии с поступающими извне тактовыми импульсами. На плате RED-V этот источник тактовых импульсов представляет собой генератор 32,768 кГц (обычно 2^{15} Гц). На **рис. 9.16** показана карта адресов памяти системного таймера. Он расположен в контроллере внутренних прерываний ядра (core-local interruptor, CLINT). Регистр `mtime` содержит 64-битное текущее значение счетчика. Его значение можно считывать или записывать;

Таблица 9.7 Регистры системного таймера

	...
0x0200BFFC	mtime (hi)
0x0200BFF8	mtime (lo)
	...
0x02004004	mtimecmp (hi)
0x02004000	mtimecmp (lo)
	...
0x02000000	msip
	...

Перепечатано из табл. 24 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

например, чтобы перезапустить таймер, можно записать в него ноль. Регистр `mtimecmp` содержит 64-битное значение для сравнения с таймером, а `msip` – регистр программных прерываний внутреннего уровня. Когда счетчик достигает значения, совпадающего с числом, записанным в `mtimecmp`, младший бит в регистре `msip` устанавливается в 1. Использование регистров `msip` и `mtimecmp` является эффективным способом проверить наличие задержки. В **табл. 9.7** показаны адреса памяти для этих регистров.

Если требуются дополнительные таймеры, для точного формирования интервалов времени можно воспользоваться дополнительными счетчиками модуля PWM (**раздел 9.3.7**).

Пример 9.5 МИГАЮЩИЙ СВЕТОДИОД

Разработайте программу, которая мигает светодиодным индикатором состояния на RED-V 5 раз в секунду в течение 4 секунд.

Решение Функция задержки в EasyREDVIO (**пример кода 9.7**) создает задержку на указанное количество миллисекунд, используя сравнение регистров таймера.

Пример кода 9.7 ФОРМИРОВАНИЕ ЗАДЕРЖКИ

```
#define MTIME_CLK_FREQ 32768          // Частота внешнего генератора в Гц
volatile uint64_t *mtime = (uint32_t*) 0x0200BFF8;
void delay(int ms) {
    uint64_t doneTime = *mtime + (ms*MTIME_CLK_FREQ)/1000;
    while (*mtime < doneTime);        // Ждем совпадения значений в регистрах
}
```

Светодиодный индикатор состояния на плате RED-V подключен к выводу GPIO5 (D13). Программа в **примере кода 9.8** настраивает этот вывод как выход. Затем она выключает и включает светодиод с помощью серии команд `digitalWrite` с частотой повторения 200 мс (5 Гц).

Пример кода 9.8 МИГАНИЕ ИНДИКАТОРОМ СОСТОЯНИЯ

```
#include "EasyREDVIO.h"

void main(void) {
    uint32_t i;

    pinMode(D13, OUTPUT);     // сконфигурировать вывод индикатора состояния как выход

    for(i = 0; i < 20; i++) {
        delay(100);
        digitalWrite(D13, 0); // выключить светодиод
        delay(100);
        digitalWrite(D13, 1); // включить светодиод
    }
}
```

9.3.7. Аналоговый ввод/вывод

Реальный мир является аналоговым. Многие встраиваемые системы нуждаются в аналоговых входах и выходах для взаимодействия с миром. Они используют *аналого-цифровые преобразователи* (АЦП) для преобразования аналоговых сигналов в цифровые значения и *цифроаналоговые преобразователи* (ЦАП), чтобы преобразовать цифровой сигнал в аналоговый. На **рис. 9.15** показаны условные графические обозначения для этих компонентов. Такие преобразователи характеризуются разрешением, динамическим диапазоном, частотой дискретизации и точностью. Например, АЦП может иметь $N = 12$-битное разрешение в диапазоне от V_{ref}^- до V_{ref}^+ (0–5 В) с частотой дискретизации f_s = 44 кГц и точностью ±3 младших значащих бита. Частота дискретизации также измеряется в выборках в секунду (samples per second, sps), где 1 sps = 1 Гц. Отношение между напряжением аналогового входа $Vin(t)$ и цифровой выборкой $X[n]$ можно выразить следующим образом:

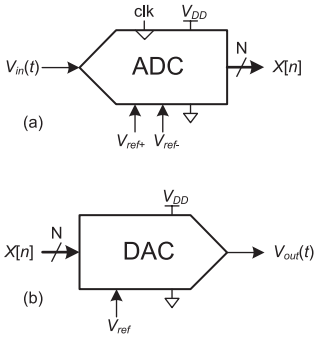


Рис. 9.15 Условные графические обозначения АЦП и ЦАП

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref}^-}{V_{ref}^+ - V_{ref}^-}. \tag{9.2}$$

Например, входное напряжение 2,5 В (половина полной шкалы) будет соответствовать $2^{12}/2$ (половина максимального значения), то есть выходное напряжение $100000000000_2 = 0x800 = 2^{11} = 2048$, с погрешностью до 3 младших значащих разрядов.

Точно так же ЦАП может иметь разрешение $N = 16$ бит в полной шкале относительно V_{ref} = 2,56 В. Выходное напряжение составляет

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref}. \tag{9.3}$$

Многие микроконтроллеры содержат встроенные АЦП с довольно умеренным быстродействием. Для достижения более высокого быстродействия (например, 16-битного разрешения или частоты дискретизации более 1 МГц) часто необходимо использовать внешний АЦП, подключенный к микроконтроллеру. Очень не многие микроконтроллеры имеют встроенные ЦАП, поэтому для преобразования цифровых значений в аналоговое напряжение тоже приходится использовать отдельные микросхемы. Тем не менее микроконтроллеры часто имитируют аналоговые выходы, используя метод, называемый *широтно-импульсной модуляцией* (ШИМ).

Цифроаналоговое преобразование

FE310 не имеет встроенного ЦАП общего назначения, поэтому в данном разделе мы рассмотрим цифроаналоговое преобразование с помощью внешних ЦАП и взаимодействие FE310 с другими микросхемами через параллельные и последовательные порты. В следующем разделе мы добьемся того же результата с использованием ШИМ.

Некоторые ЦАП принимают N-битный цифровой вход через параллельный интерфейс с N проводами, в то время как другие принимают его через последовательный интерфейс, такой как SPI. Некоторые ЦАП требуют как положительного, так и отрицательного напряжения питания, в то время как другие работают от одного источника. Некоторые ЦАП поддерживают гибкий диапазон напряжений питания, в то время как другие требуют определенного напряжения. Входные логические уровни должны быть совместимы с цифровым источником. Некоторые ЦАП дают выходное напряжение, пропорциональное цифровому входу, в то время как другие управляют токовым выходом; может понадобиться операционный усилитель для преобразования этого тока в напряжение в требуемом диапазоне.

В этом разделе мы используем 12-битный параллельный ЦАП фирмы Linear Technology LTC1450 и 8-битный последовательный ЦАП Microchip MCP4801. Обе микросхемы выдают на выходе напряжение[1], работают от одного источника питания 5–15 В, используют V_{IH} = 2,4 В для совместимости с логическими уровнями 3,3 В и поставляются в DIP-корпусах, что упрощает их монтаж и использование. LTC1450 формирует выходной сигнал в диапазоне от 0 до 2,048 В или от 0 до 4,095 В в зависимости от настройки усиления, потребляет 2 мВт, поставляется в корпусе с 24 выводами и имеет время установки 4 мкс, что обеспечивает выходную скорость 250 тыс. отсчетов в секунду. Техническое описание доступно на сайте http://analog.com. Микросхема MCP4801 формирует выходной сигнал в диапазоне 0–2,048 В или 0–4,096 В, потребляет менее 2 мВт, поставляется в корпусе с 8 выводами и имеет время установки 4,5 мкс. Порт SPI этой микросхемы работает на максимальной частоте 20 МГц. Техническое описание доступно на http://microchip.com.

Пример 9.6 АНАЛОГОВЫЙ ВЫВОД ЧЕРЕЗ ВНЕШНИЙ ЦАП

Разработайте схему и программу простого генератора сигналов, генерирующего синусоидальные и треугольные волны с использованием RED-V, LTC1450 и MCP4801.

Решение Схема изображена на **рис. 9.16**. В этом примере используются две микросхемы ЦАП. Оба ЦАП запитаны от источника питания 5 В и шунтированы конденсаторами 0,1 мкФ для снижения помех в цепях питания.

[1] Существуют ЦАП с токовым выходом, применяемые в основном в прецизионных и высокочастотных схемах. – *Прим. перев.*

ЦАП LTC1450 имеет 12-разрядный вход D0–D11, который определяет аналоговое напряжение, поступающее на выход VOUT. В нашем примере мы используем только 8-битную точность, поэтому привязываем четыре младших бита (D0–D3) к «земле». Чтобы загрузить данные в ЦАП, RED-V сначала задает желаемое значение на линиях данных D4–D11, а затем устанавливает сигнал низкого уровня на выводе записи с активным низким уровнем (WR) для записи данных в ЦАП. Вывод CLR подключен к VCC, потому что нам не нужно очищать регистры входных данных. Вывод LDAC с активным низким уровнем постоянно подключен к земле, чтобы загрузка данных происходила при каждом переходе WR в низкий уровень.

ЦАП MCP4801 подключается к RED-V через SPI1. Помимо стандартных сигналов SPI, MCP4801 имеет вывод аналогового выходного напряжения (VOUT) и два входа управления с активным низким уровнем: аппаратное отключение (SHDN) и ЦАП с защелкой (LDAC). SHDN используется для отключения схемы управления выходом и экономии энергии, когда выходное напряжение не требуется. Низкий уровень на входе LDAC разрешает запись данных в ЦАП. Для записи данных в MCP4801 через SPI контроллер передает 16-битное значение: биты с 11 по 4 содержат данные D7–D0; бит 13 – выбор усиления (однократное, если установлен в 1, и двукратное, если установлен в 0); а бит 12 определяет состояние SHDN (0 – отключает выход, 1 – разрешает вывести на VOUT напряжение). В данном случае SHDN управляется программно, поэтому на схеме он остается не подключенным к внешним цепям.

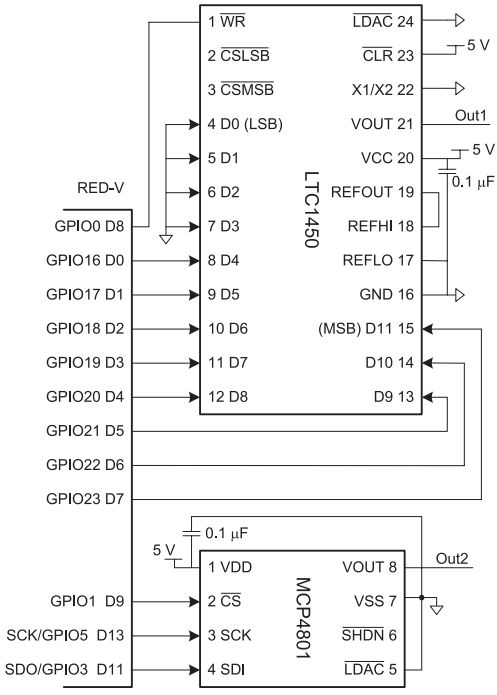


Рис. 9.16 Подключение ЦАП к плате RED-V с помощью параллельного и последовательного интерфейсов

Программа для управления обоими ЦАП представлена в **примере кода 9.9**. Программа настраивает 8 выводов параллельного порта как выходы, а также настраивает GPIO0 как выход для управления сигналом WR на LTC1450 и GPIO1 для управления сигналом выбора микросхемы на MCP4801. Она инициализирует SPI для работы на частоте до

Пример кода 9.9 ГЕНЕРАЦИЯ СИНУСОИДАЛЬНОГО ИМПУЛЬСА С ПОМОЩЬЮ ЦАП

```
#include "EasyREDVIO.h"
#include <math.h> // библиотека требуется для использования функции синуса

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];

#define SHDNn_Pos 12
#define Gain_Pos 13

int parallelPins[8] = {D0, D1, D2, D3, D4, D5, D6, D7};

void initWaveTables(void) {
  int i;
  for (i = 0; i<NUMPTS; i++) {
    sine[i] = 127*(sin(2*3.14159*i/NUMPTS) + 1);  // разрядность 8 бит
    if (i < NUMPTS/2) triangle[i] = i*255/NUMPTS; // разрядность 8 бит
    else triangle[i] = 254 - i*255/NUMPTS;
  }
}

void genWaves(int freq) {
  int i, j;
  int delay_cycles = MTIME_CLK_FREQ/(NUMPTS*freq);

  for (i = 0; i < 2000; i++){
    for (j = 0; j < NUMPTS; j++) {
      uint64_t doneTime = *mtime + delay_cycles; // Период выборки

      // Load serial DAC
      digitalWrite(1, 0);  // выбрать чип (CS = 0)
      // Сделать SHDNn активным (бит 12) и единичное усиление (бит 13)
      volatile uint16_t sine_samp_dac = ((uint16_t) sine[j] << 4) \
                                        |(1 << SHDNn_Pos) | (1 << Gain_Pos);
      spiSendReceive16(sine_samp_dac);
      digitalWrite(1, 1);       // отключить выбор чипа (CS = 1)

      // Загрузка параллельного ЦАП
      digitalWrite(0, 1);       // Не загружать, пока меняются данные
      digitalWrites(parallelPins, 8, triangle[j]);
      digitalWrite(0, 0);       // Загрузить новую выборку в ЦАП
      while(*mtime < doneTime); // Ждем mtime_cmp для отправки
    }
  }
}

int main(void) {
  pinsMode(parallelPins, 8, OUTPUT); // Настроить выводы, подключенные к AD558 как выходы
  pinMode(0, OUTPUT);                // Настроить вывод 0 как выход для управления линией LOAD
  pinMode(1, OUTPUT);                // настроить вывод 1 как выход для управления линией CE
  spiInit(15, 0, 0);  // Инициализация SPI
  initWaveTables();
  genWaves(100);
}
```

500 кГц. Функция `initWaveTables` предварительно вычисляет массив значений выборок для синусоидальных и треугольных волн. Затем она обновляет последовательный ЦАП. Далее программа приостанавливается до тех пор, пока таймер не покажет, что пришло время для следующей выборки. Максимальная частота генерируемых сигналов устанавливается временем отправки каждой точки в функции `genWaves`, которое ограничено временем передачи SPI.

Широтно-импульсная модуляция

Другим способом генерировать аналоговый выходной сигнал в цифровой системе является широтно-импульсная модуляция (ШИМ), в которой периодический выходной сигнал принимает высокое значение в течение части периода передачи и низкое в оставшейся части. Доля периода, в которой сигнал имеет высокое значение, называется *коэффициентом заполнения*, или *скважностью* (duty cycle), как показано на **рис. 9.17**. Среднее значение напряжения на выходе пропорционально коэффициенту заполнения. Например, если значения низкого и высокого уровней составляют 0 В и 3,3 В соответственно, а коэффициент заполнения – 25 %, то среднее значение напряжения будет 0,25×3,3 = 0,825 В. Низкочастотная фильтрация сигнала ШИМ устраняет пульсации, и выходной сигнал принимает требуемое среднее значение. Таким образом, ШИМ является эффективным способом реализации аналогового выхода, если частота следования импульсов намного выше, чем частота желаемого аналогового сигнала. К другим применениям ШИМ относятся формирование прямоугольных импульсов звуковой частоты и цифровое управление мощностью двигателя или яркостью источника света.

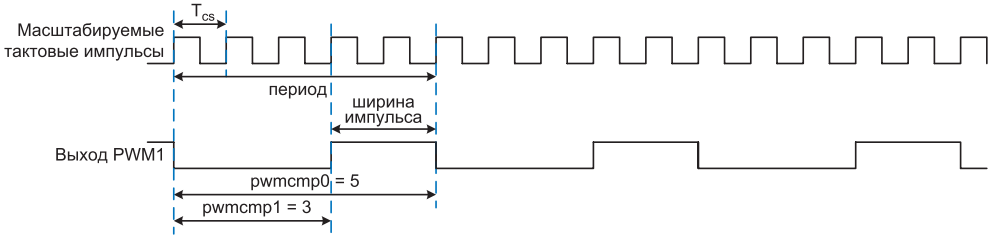


Рис. 9.17 Сигнал с широтно-импульсной модуляцией (ШИМ)

FE310 имеет три периферийных модуля ШИМ, и, как показано в **табл. 9.3**, каждый модуль имеет четыре выхода ШИМ, т. е. доступно 12 выходов ШИМ. Выходы модуля PWM0 имеют 8-битную точность, а PWM1 и PWM2 – 16-битную точность. В этом разделе мы покажем, как использовать PWM2, но при настройке и использовании двух других модулей PWM выполняются аналогичные шаги. PWM2 имеет четыре выхода (`PWM2_PWM0`, `PWM2_ PWM1`, `PWM2_PWM2`, `PWM2_PWM3`), которые доступны на контактах GPIO10–13 с использованием функции контактов `IOF1`.

Таблица 9.8 Регистры конфигурации PWM2

...	...
0x1002502C	pwmcmp3
0x10025028	pwmcmp2
0x10025024	pwmcmp1
0x10025020	pwmcmp0
...	...
0x10025010	pwms
0x1002500C	...
0x10025008	pwmcount
0x10025004	...
0x10025000	pwmcfg
...	...

Перепечатано из табл. 89 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

ШИМ может использоваться в нескольких режимах генерации, но мы рассмотрим генерацию сигналов ШИМ, аналогичных представленным на **рис. 9.17**. Для этого модуль настраивают для работы в циклическом режиме, в котором компаратор 0 (pwmcmp0) устанавливает период, а компаратор 1 (pwmcmp1) устанавливает минимальное время. Это время выражается в единицах периода тактовой частоты T_{cs} после делителя. Например, как показано на **рис. 9.17**, если период тактовой частоты составляет 0,5 мкс (2 МГц) и pwmcmp0 = 5, то на выходе PWM2_PWM1 (вывод 11) будут присутствовать импульсы с периодом 5 × 0,5 мкс = 2,5 мкс (400 кГц). Если pwmcmp1 = 3, то коэффициент заполнения составляет 1 – (3/5) = 40 %.

В **табл. 9.8** представлена карта адресов памяти для регистров PWM2. В этом разделе мы описываем шаги, необходимые для настройки вывода PWM1_PWM1; остальные ШИМ и их выходы настраиваются аналогичным образом.

В **табл. 9.9** представлены битовые поля регистра конфигурации ШИМ pwmcfg. Обратите внимание, что большинство битов не устанавливаются в ноль при сбросе системы, поэтому рекомендуется начинать программу с обнуления всех битов, а затем записи 1 в поля pwmenalways и pwmzerocmp, чтобы настроить ШИМ для генерации непрерывной последовательности импульсов с периодом, заданным в pwmcmp0.

Тактовая частота после делителя f_{scaled} – это тактовая частота базовой шины f_{base} = 16 МГц, поделенная на 2^{pwmscale}, где pwmscale – это 4-битное число в диапазоне от 0 до 15, сохраненное в регистре pwmcfg. Частота ШИМ равна $f_{pwm} = f_{scaled} / \text{pwmcmp0} = f_{base} / (\text{pwmcmp0} \times 2^{\text{pwmscale}})$. Как сказано выше, коэффициент заполнения равен 1 – (pwmcmp1/pwmcmp0). Существует множество возможных сочетаний pwmscale и pwmcmp0, что позволяет получить желаемую частоту PMW. При этом разрешение по частоте ШИМ (минимальная разница между желаемой и фактической частотами) является наилучшим, когда pwmscale как можно меньше, а pwmcmp0 как можно больше, при условии что pwmcmp0 является 16-битным числом без знака (т. е. не может превышать 65 535).

Пример 9.7 ШИРОТНО-ИМПУЛЬСНАЯ МОДУЛЯЦИЯ (ШИМ)

Выберите значения pwmscale и pwmcmp0 такие, чтобы светодиод мигал с частотой 1,2 Гц. Повторите подбор значений, чтобы сгенерировать тон с частотой 1190 Гц.

Решение Предположим, что мы хотим мигать светодиодом с частотой f_{pwm} = 1,2 Гц. $f_{pwm} = f_{base}/(\text{pwmcmp0} \times 2^{\text{pwmscale}})$. Выберем pwmscale = 8 и pwmcmp0 = 52 083,33, чтобы получить желаемую частоту с f_{scaled} = 16 МГц / 28 = 62,5 кГц. pwmcmp0 – это 16-битный регистр, поэтому мы должны округлить полученное число до 52 083, что дает фактическое значение f_{pwm} = 16 МГц / (52 083 × 2^8) =

1,20000768 Гц, что очень близко к желаемой частоте и сравнимо с точностью 10 миллионных долей, присущей типичным кварцевым часам.

Теперь предположим, что нам нужен выходной сигнал с частотой 1190 Гц. Если не менять значение `pwmscale`, то `pwmcmp0` должно быть равно 52,521. Округление до 53 дает фактическое значение f_{pwm} = 16 МГц / (53×28) = 1179,2 Гц, т. е. ошибку около 10 Гц, или 0,91 %. Если нам нужна более высокая точность, мы могли бы уменьшить `pwmscale` до нуля и увеличить `pwmcmp0` до 13 445, получив f_{pwm} = 16 МГц / (13 445×20) = 1190,03 Гц.

Таблица 9.9 Поля регистра конфигурации ШИМ

Регистр конфигурации ШИМ (`pwmcfg`)				
Смещение адреса		`0x0`		
Биты	**Имя поля**	**Атрибут**	**Сброс**	**Описание**
[3:0]	`pwmscale`	`RW`	`X`	Коэффициент деления счетчика ШИМ
[7:4]	Зарезервировано			
8	`pwmsticky`	`RW`	`X`	Запрет обнуления битов `pwmcmpXip`
9	`pwmzerocmp`	`RW`	`X`	При совпадении счетчик устанавливается в ноль
10	`pwmdeglitch`	`RW`	`X`	Хранить `pwmcmpXip` на протяжении такта
11	Зарезервировано			
12	`pwmenalways`	`RW`	`0x0`	ШИМ работает непрерывно
13	`pwmenoneshot`	`RW`	`0x0`	Запустить один цикл генерации ШИМ
[15:14]	Зарезервировано			
16	`pwmcmp0center`	`RW`	`X`	Центральное значение компаратора PWM0
17	`pwmcmp1center`	`RW`	`X`	Центральное значение компаратора PWM1
18	`pwmcmp2center`	`RW`	`X`	Центральное значение компаратора PWM2
19	`pwmcmp3center`	`RW`	`X`	Центральное значение компаратора PWM3
[23:20]	Зарезервировано			
24	`pwmcmp0gang`	`RW`	`X`	Совпадение PWM0/PWM1
25	`pwmcmp1gang`	`RW`	`X`	Совпадение PWM1/PWM2
26	`pwmcmp2gang`	`RW`	`X`	Совпадение PWM2/PWM3
27	`pwmcmp3gang`	`RW`	`X`	Совпадение PWM3/PWM0
28	`pwmcmp0ip`	`RW`	`X`	Запрос прерывания PWM0
29	`pwmcmp1ip`	`RW`	`X`	Запрос прерывания PWM1
30	`pwmcmp2ip`	`RW`	`X`	Запрос прерывания PWM2
31	`pwmcmp3ip`	`RW`	`X`	Запрос прерывания PWM3

Перепечатано из табл. 91 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Драйвер устройства PWM может иметь функции `pwmInit()` и `pwm(int freq, float duty)`. Функция `pwmInit` настроит соответствующий вывод для работы периферийного устройства PWM и установит биты в регистре `pwmcfg`. Функция `pwm` задаст соответствующие значения `pwmscale`, `pwmcmp0` и `pwmcmp1` для генерации сигнала с заданной частотой и коэффициентом заполнения. Разработка этих функ-

ций аналогична разработке драйвера устройства SPI или UART; детали мы оставим в качестве упражнения для читателя.

Аналого-цифровое преобразование

Многие микроконтроллеры имеют хотя бы один встроенный АЦП, но в FE310 его нет. В этом разделе описывается аналого-цифровое преобразование с использованием внешнего преобразователя, аналогичного внешним ЦАП, описанным в предыдущем разделе.

Пример 9.8 АНАЛОГОВЫЙ ВХОД С ВНЕШНИМ АЦП

Подключите 10-битный аналого-цифровой преобразователь MCP3002 к FE310 с помощью SPI и выведите в консоль значение входного напряжения. В качестве эталонного напряжения используйте полное напряжение питания 3,3 В. Детальное описание микросхемы АЦП самостоятельно найдите в интернете.

Решение На **рис. 9.18** показана схема соединений между FE310 и АЦП MCP3002, а в **примере кода 9.10** показан код драйвера. MCP3002 использует напряжение питания в качестве эталона: вывод VDD (8) подключен к 3,3 В. На этот вывод можно подать напряжение от 3,3 до 5,5 В; мы выбираем 3,3 В. АЦП имеет два входных канала, CH0 и CH1. Мы подключаем канал 0 к потенциометру и вращением его рукоятки выбираем входное напряжение между 0 и 3,3 В.

В **примере кода 9.10** контроллер FE310 инициализирует SPI, после чего многократно считывает и выводит в консоль значение измеренного напряжения. В соответствии с техническим описанием FE310 должен отправить 16-битное число 0x6000 через SPI для чтения CH0 и получить обратно 10-битный результат в младших 10 битах 16-битного результата. Поскольку мы не можем напрямую настроить FE310 на передачу 16-битных пакетов, мы можем передать два 8-битных пакета подряд, не меняя состояние вывода выбора чипа между ними. Хотя встроенный модуль SPI способен автоматически управлять линией выбора микросхемы, здесь мы вручную настраиваем GPIO2 как выход и переключаем его соответствующим образом в начале передачи (записывая ноль) и в конце передачи (записывая единицу).

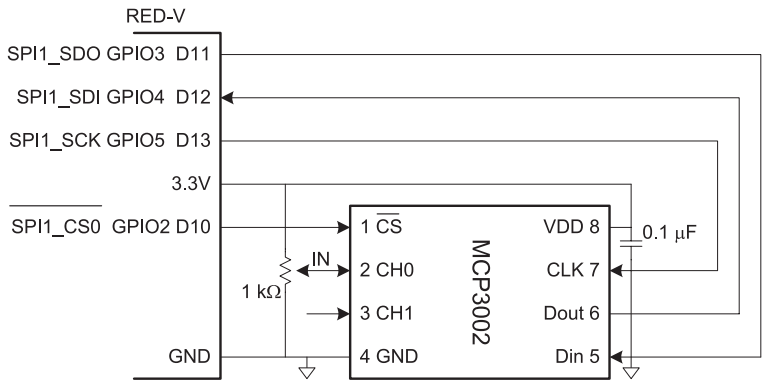


Рис. 9.18 Считывание аналогового входа с помощью внешнего АЦП

Пример кода 9.10 КОД ДЛЯ ВЗАИМОДЕЙСТВИЯ С АЦП

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t sample;
  spiInit(15, 0, 0); // Инициализируем SPI
                     // Делитель частоты div = 15, CPOL = 0, CPHA = 0
  pinMode(D10, OUTPUT);
  while(1) {
    digitalWrite(D10, 0);
    spiSendReceive('0x60');
    sample = spiSendReceive('0x00');
    digitalWrite(D10, 1);
    printf("Read %d\n", sample);
    delay(200);
  }
}
```

9.3.8. Прерывания

До сих пор мы полагались на циклический *опрос*, при котором программа постоянно проверяет некоторое значение до тех пор, пока не произойдет событие, такое как поступление данных на UART или достижение таймером значения сравнения. Подобный подход – это пустая трата вычислительной мощности процессора, и вдобавок он затрудняет разработку программ, выполняющих полезную работу в ожидании возникновения событий.

Большинство микроконтроллеров поддерживают *прерывания*. Когда происходит событие, микроконтроллер останавливает выполнение основной программы и переходит к *обработчику прерывания*, который реагирует на прерывание. Завершив обработку прерывания, процессор возвращается к основной программе и аккуратно продолжает работу точно с того места, где она была прервана. Прерывания – это аппаратные исключения, которые обсуждались в **разделе 6.6.2**.

FE310 содержит контроллер внутренних прерываний ядра (core-local interruptor, CLINT), который обрабатывает прерывания от таймера и программные прерывания. Программные прерывания используются для межпроцессорной связи и отладки. FE310 также имеет контроллер внешних прерываний (platform-level interrupt controller, PLIC), который собирает прерывания от других периферийных устройств. В многопроцессорной системе PLIC направляет внешнее прерывание соответствующему процессору для обработки.

В **примере 9.9** мы рассмотрим разработку программы управления светодиодом с использованием прерывания от таймера вместо опроса.

Пример 9.9 УПРАВЛЕНИЕ СВЕТОДИОДОМ С ПОМОЩЬЮ ПРЕРЫВАНИЯ ОТ ТАЙМЕРА

Мы настраиваем локальные прерывания на FE310 с помощью CLINT. Для микросхемы FE310-G002, используемой в платах RED-V RedBoard и RED-V Thing Plus, информация об использовании прерываний представлена в главах 8–10 руководства FE310-G002. Базовая процедура настройки локальных прерываний через CLINT описана ниже.

1. Разработать обработчик прерывания, который будет срабатывать всякий раз, когда запускается прерывание или исключение. Основное назначение обработчика прерывания – выяснить, какое прерывание или исключение было инициировано, а затем выполнить необходимые действия.
2. Настроить регистр управления и состояния (control and status register, CSR) `mtvec`, указав адрес обработчика прерываний и режим (прямой или векторный).
3. Разрешить конкретное прерывание (например, от таймера).
4. Установить глобальное разрешение прерываний.

За определением констант и массивов указателей функций в коде следует объявление функции обработчика глобальной ловушки `handle_trap()`, как показано в **примере кода 9.11**. Эта функция выполняется всякий раз, когда срабатывает ловушка (случается прерывание или исключение). Ее задача – выяснить, какое событие послужило источником вызова, и перейти к правильному обработчику прерывания или исключения. Обработчик прерываний выполняет две задачи. Во-первых, он использует маску (`MCAUSE_INT_MASK`) для проверки самого старшего бита регистра `mcause`, который указывает, с каким событием мы имеем дело: с прерыванием (сгенерированным внешним по отношению к ядру устройством) или исключением (сгенерированным внутри ядра). Структура регистра `mcause` показана в **табл. 9.10**, а список кодов прерываний и исключений – в **табл. 6.6**. Потом обработчик использует дополнительную маску (`MCAUSE_CODE_ MASK`) для определения кода прерывания и переходит к соответствующему обработчику прерывания или исключения на основе индекса массивов указателей функций `interrupt_handler` или `exception_handler`.

Затем мы определяем *процедуру обслуживания прерывания* (interrupt service routine, ISR) для таймера. Это функция, которая выполняет определенные команды всякий раз, когда процессор получает прерывание от таймера. В данном примере мы дадим этой функции имя `timer_handler()`. Она считывает текущее состояние вывода GPIO, управляющего встроенным светодиодом (D1/GPIO5), и переключает состояние на противоположное с помощью `digitalWrite()`.

Таблица 9.10 Поля регистра `mcause`

Биты	Имя поля	Описание
[9:0]	Код исключения	Код, обозначающий самое последнее исключение
[30:10]	Зарезервировано	
31	Прерывание	1, если событие прерывания; 0 в ином случае

Перепечатано из табл. 22 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Пример кода 9.11 НАСТРОЙКА ОБРАБОТЧИКОВ ПРЕРЫВАНИЙ

```
// Массивы указателей функций для обработчиков прерываний и исключений
#define MAX_INTERRUPTS 16
void (*interrupt_handler[MAX_INTERRUPTS])();
void (*exception_handler[MAX_INTERRUPTS])();

// Маски для определения прерываний и исключений и соответствующий код
#define MCAUSE_INT_MASK 0x80000000   // Если [31] = 1, то прерывание, иначе исключение
#define MCAUSE_CODE_MASK 0x7FFFFFFF // в младших битах представлен код

// Объявление обработчика прерываний. Объявлен с атрибутом, который
// указывает на функцию-хелпер GCC.
void handle_trap(void) __attribute((interrupt));

// Объявление обработчика ловушки
void handle_trap() {
    unsigned long mcause_value = read_csr(mcause);
    if (mcause_value & MCAUSE_INT_MASK) {
        // Переход на обработчик прерываний
        // Индексы 32-битного массива содержат адреса функций
        interrupt_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
    else {
       // Переход на обработчик исключений
       exception_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
}
```

Затем функция сбрасывает таймер, вызывая другую функцию `reset_timer()`, которая устанавливает текущий счетчик в регистре `mtime` на 0 и сбрасывает значение счетчика, при котором должно быть запущено следующее прерывание.

Пример кода 9.12 ТАЙМЕР ISR И ФУНКЦИЯ СБРОСА ТАЙМЕРА

```
void timer_handler() {
    volatile int pin_val = (GPIO0->output_val >> D13) & 1; // Чтение текущего
                                                           // состояния выхода
    if(pin_val) digitalWrite(D13, LOW);
    else digitalWrite(D13, HIGH);
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));
}

void reset_timer(int count_val) {
    *MTIME = 0;
    *MTIMECMP = count_val;
}
```

В отличие от других регистров, которые мы использовали в этой главе, большинство регистров, связанных с CLINT, таких как `mtvec`, `mie` и `mstatus`, не отображены в память. Эти регистры называются *регистрами управления и состояния* (CSR). Чтобы управлять такими регистрами, мы должны использовать инструкции ассемблера RISC-V: чтение CSR (`csrr`) и запись CSR (`csrw`). Эти инструкции можно обернуть в макросы C, чтобы облегчить взаимодействие с ними.

Пример кода 9.13 МАКРОСЫ ДЛЯ ЗАПИСИ И ЧТЕНИЯ CSR

```
// Макросы для чтения и записи регистров управления и состояния (CSRs)
#define read_csr(reg) ({ unsigned long __tmp; \
  asm volatile ("csrr %0, " #reg : " = r"(__tmp)); \
  __tmp; })

#define write_csr(reg, val) ({ \
  asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })
```

После настройки обработчика ловушек мы регистрируем его, помещая адрес в регистр `mtvec`, структура которого представлена в **табл. 9.11**. `mtvec` – это 32-битный регистр, в котором биты [31:2] содержат соответствующие биты [31:2] адреса функции обработчика прерываний (биты [1:0] адреса автоматически считаются равными нулю, поскольку инструкции должны быть выровнены в памяти по словам). Вместо адреса биты [1:0] регистра `mtvec` используются для настройки обработки исключений в прямом или векторном режиме. В прямом режиме, независимо от того, какое прерывание или исключение возникло, мы переходим к адресу функции, указанному в битах `mtvec` [31:2]. Этот режим мы используем в данном примере. В векторном режиме мы переходим к разным адресам памяти в зависимости от прерывания.

Таблица 9.11 **Поля регистра `mtvec`**

Биты	Имя поля	Описание
[1:0]	MODE	Режим обработки прерываний: прямой (`00`) или векторный (`10`)
[31:2]	BASE[31:2]	Базовый адрес обработчика `trap_handler`

Перепечатано из табл. 18 руководства SiFive FE310-G0002, с разрешения © SiFive, Inc., 2019.

Выполнив настройку обработчика прерываний и процедуры обслуживания прерывания таймера, мы завершаем подготовку тем, что разрешаем прерывание от таймера. Для этого мы устанавливаем бит 7 разрешения прерывания таймера (MTIE) в регистре разрешения прерывания `mie` и разрешаем прерывания глобально, устанавливая бит 3 разрешения прерывания (MIE) в регистре состояния процессора (`mstatus`). Простые вспомогательные функции для глобального включения и отключения прерываний показаны в **примере кода 9.15**. Полную информацию о структуре регистров `mstatus` и `mie` можно найти в табл. 17 и 20 технического описания FE310-G002 от компании SiFive.

Наконец, мы вызываем функции, которые объявили в нашей главной функции, как показано в **примере кода 9.16**. В данном случае, поскольку наше приложение управляется прерываниями, мы ничего не делаем в основном цикле `while`.

Следует проявлять осторожность при разработке приложений, критических с точки зрения безопасности или точного времени выполнения,

поскольку прерывания являются асинхронными событиями и могут случиться в любой момент во время выполнения программы. Вы как программист должны подумать о том, какие ошибки могут быть вызваны срабатыванием прерывания в неподходящее время. Если у вас есть сегмент кода, при выполнении которого вы хотите избежать прерывания, вы можете отключить прерывания (т. е. сбросить бит `MIE` в регистре `mstatus`) при выполнении критически важных команд, а затем повторно включить прерывания по завершении (т. е. установить `MIE` в `mstatus`).

Пример кода 9.14 РЕГИСТРАЦИЯ ОБРАБОТЧИКА ЛОВУШЕК ПУТЕМ ЗАПИСИ В `mtvec`

```
void register_trap_handler(void *func) {
    // Записать в разряды mtvec[31:2] адрес функции обработчика прерывания
    // Два самых младших значащих разряда игнорируем, т. к. адреса команд
    // в памяти всегда кратны 4 байтам
    // Записываем в mtvec[1:0] значение 00 (прямой режим).
    write_csr(mtvec, ((unsigned long) func) & ~(0b11));
}
```

Пример кода 9.15 ГЛОБАЛЬНОЕ ВКЛЮЧЕНИЕ И ОТКЛЮЧЕНИЕ ПРЕРЫВАНИЙ

```
void enable_interrupts() {
    // Установить в 1 бит 3 в mstatus (MIE), чтобы разрешить прерывания
    write_csr(mstatus, read_csr(mstatus) | (1 << 3));

void disable_interrupts() {
    // Сбросить в 0 бит 3 в mstatus (MIE), чтобы запретить прерывания
    write_csr(mstatus, read_csr(mstatus) & ~(1 << 3));
}
```

Пример кода 9.16 УПРАВЛЕНИЕ СВЕТОДИОДОМ ПРИ ПОМОЩИ ПРЕРЫВАНИЙ ТАЙМЕРА

```
#include "EasyREDVIO.h"

// Указатели карты памяти CLINT
#define MTIMECMP ((uint64_t *) 0x02004000UL)
#define MTIME ((uint64_t *) 0x0200BFF8UL)

#define BLINK_FREQ 4 // Произвольная константа, определяющая частоту мигания
#frequency

int main(void) {

    // Настраиваем вывод светодиода как выход
    pinMode(D13, OUTPUT);

    // Регистрируем обработчик прерывания.
    // Прерыванию таймера соответствует код исключения 7, поэтому
    // функция timer_handler() является элементом массива с индексом 7.
    interrupt_handler[7] = timer_handler;
```

Пример кода 9.16 (окончание)

```
    // Настраиваем регистр mtvec
    register_trap_handler(handle_trap);

    // Сбрасываем таймер
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));

    // Разрешаем прерывание от таймера
    write_csr(mie, read_csr(mie) | (1 << 7));

    enable_interrupts();

    while(1) {
    };
    return 0;
}
```

9.4. Другие внешние устройства микроконтроллера

Микроконтроллеры часто взаимодействуют с другими внешними устройствами. В этом разделе приведены примеры различных распространенных внешних устройств, включая символьные жидкокристаллические дисплеи (LCD), мониторы VGA, модули беспроводной связи Bluetooth и контроллеры двигателей.

9.4.1. Символьные ЖК-дисплеи

Символьный ЖК-дисплей – это небольшой жидкокристаллический дисплей, предназначенный для отображения одной или нескольких строк текста. Они широко используются в лицевых панелях таких устройств, как кассовые аппараты, лазерные принтеры и факсы, которым необходимо отображать ограниченный объем информации. Они легко взаимодействуют с микроконтроллером через параллельный интерфейс, интерфейс RS-232 или SPI. Компания Crystalfontz America продает широкий спектр символьных ЖК-дисплеев, начиная от форм-фактора в 8 столбцов × 1 строку до 40 столбцов × 4 строки с выбором цвета, подсветки, питанием 3,3/5 В и видимостью при дневном свете. Эти ЖК-дисплеи могут стоить $20 или более в розницу и менее $5 при больших объемах покупки.

В этом разделе показано, как подключить плату RED-V к параллельному ЖК-дисплею Crystalfontz CFAH2002A-TMI-JT 20×2, представленному на **рис. 9.19**. Передача данных происходит по параллельному интерфейсу, совместимому с ЖК-контроллером HD44780, который разработан Hitachi и фактически стал промышленным стандартом.

Рис. 9.19 ЖК-дисплей CFAH2002A-TMI 20 × 2 фирмы Crystalfontz
(напечатано с разрешения Crystalfontz America©, 2012)

На **рис. 9.20** показан ЖК-дисплей, подключенный к плате RED-V через 8-битный параллельный интерфейс (входы D0–D7 на ЖК-дисплее). ЖК-дисплей работает при 5 В, но совместим с логическими уровнями 3,3 В платы RED-V. Контрастность ЖК-дисплея устанавливается вторым напряжением (вход 3, VO), формируемым с помощью потенциометра. Как правило, экран наиболее комфортно читается при диапазоне напряжений 4,2–4,8 В. На ЖК-дисплей поступают три сигнала управления: RS (1 – для символов, 0 – для инструкций), R/$\overline{W}$ (1 – для чтения с дисплея, 0 – для записи) и E (импульс высокого уровня длительностью не менее 250 нс для включения ЖК-дисплея, когда следующий байт данных будет готов к передаче). Помимо передачи битов данных, линии данных D0–D7 используются для настройки конфигурации ЖК-дисплея, когда RS = 0 (т. е. в режиме команд). При чтении порт ЖК-дисплея D7 возвращает флаг занятости, который равен единице, когда ЖК-дисплей занят, и нулю, когда он готов принять другую инструкцию или байт данных.

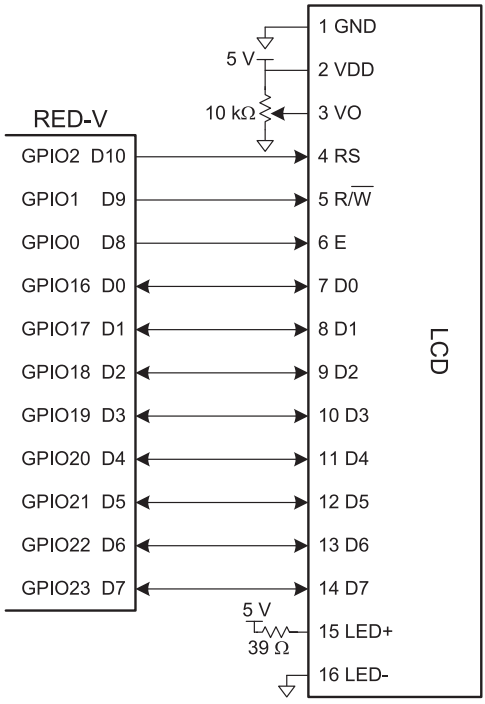


Рис. 9.20 Подключение ЖК-дисплея с помощью параллельного интерфейса

Для инициализации ЖК-дисплея плата RED-V должна отправить в него последовательность команд, показанных в **табл. 9.12**. Для записи команд в дисплей необходимо установить на линиях RS и R/$\overline{W}$ нулевые значения, вывести код команды на восемь линий данных и сгенерировать импульс высокого уровня на линии E длительностью не менее 250 нс. Байты данных записываются аналогичным образом, за исключением того, что RS = 1. После отправки команды или байта данных процессор должен подождать некоторое время (или пока не будет сброшен флаг занятости) перед отправкой другой команды или байта данных. Для считы-

вания флага занятости (D7) необходимо установить логические уровни RS = 0 и R/$\overline{W}$ = 1 и сгенерировать импульс на линии E длительностью не менее 250 нс. Перед чтением флага занятости (D7) необходимо временно настроить вывод GPIO23 как вход.

Таблица 9.12 Последовательность инициализации ЖК-дисплея

Код (D7-D0)	Назначение	Ожидание, мкс
(подача питания)	Включение дисплея	15 000
0x30	Установить режим 8 бит	4100
0x30	Снова установить режим 8 бит	100
0x30	Еще раз установить режим 8 бит	Пока есть флаг занятости
0x3C	2 строки, символы 5×8 точек	Пока есть флаг занятости
0x08	Выключить дисплей	Пока есть флаг занятости
0x01	Очистить дисплей	1530
0x06	Режим ввода символов со смещением курсора после каждого символа	Пока есть флаг занятости
0x0C	Включить дисплей без курсора	

(В таблице приведены коды инструкций, поэтому RS = 0 и $\mathbf{R/\overline{W}}$ = 0.)

После завершения процедуры настройки ЖК-дисплей готов принять текст для отображения. Для записи текста в ЖК-дисплей необходимо установить уровни RS = 1 и R/$\overline{W}$ = 0, поместить байт данных на линии D0–D7 и сформировать на линии E единичный импульс длительностью не менее 250 нс. После передачи каждого символа плата RED-V должна дождаться сброса бита занятости и лишь потом отправлять следующий символ. Она также может отправить команду 0x01 для очистки дисплея или 0x02 для возврата курсора в исходное положение в верхнем левом углу.

Пример 9.10 УПРАВЛЕНИЕ ЖК-ДИСПЛЕЕМ

Напишите программу для вывода строки «I love LCDs!» на символьном дисплее Crystalfontz CFAH2002A-TMI.

Решение В **примере кода 9.17** показан код программы, которая инициализирует дисплей, а затем посимвольно выводит на него строку «I love LCDs!».

Пример кода 9.17 ВЫВОД СТРОКИ «I LOVE LCDs» НА ЖК-ДИСПЛЕЙ

```
#include "EasyREDVIO.h"

int LCD_IO_Pins[] = {D0, D1, D2, D3, D4, D5, D6, D7};

typedef enum {INSTR, DATA} mode;
#define RS D10
#define RW D9
#define E D8
```

Пример кода 9.17 (окончание)

```
char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md = = DATA));   // установить режим передачи данных
    digitalWrite(RW, 1);              // RW = режим чтения
    digitalWrite(E, 1);               // высокий уровень на линии E
    delay(1);                         // ожидание ответа дисплея
    c = digitalReads(LCD_IO_Pins, 8); // чтение байта через параллельный порт
    digitalWrite(E, 0);               // низкий уровень на линии E
    delay(1);
    return c;
}

void lcdBusyWait(void) {
    char state;
    do {
        state = lcdRead(INSTR);
    } while(state & 0x80);
}

void lcdWrite(char val, mode md) {
    pinsMode(LCD_IO_Pins, 8, OUTPUT);
    digitalWrite(RS, (md = = DATA));    // режим передачи данных, OUTPUT = 1, INPUT = 0
    digitalWrite(RW, 0);                // RW = режим записи (RW = 0)
    digitalWrites(LCD_IO_Pins, 8, val); // запись байта в параллельный порт
    digitalWrite(E, 1); delay(1);       // короткий положительный импульс на линии E
    digitalWrite(E, 0); delay(1);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delay(1);
}

void lcdPrintString(char* str) {
    while (*str ! = 0) {
        lcdWrite(*str, DATA); lcdBusyWait();
        str++;
    }
}

void lcdInit(void) {
    pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
    // последовательность команд для инициализации:
    delay(15);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

int main(void) {
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

9.4.2. VGA-монитор

Более гибкий вариант отображения данных – использование компьютерного монитора. В этом разделе показано, как работать с монитором стандарта VGA (Video Graphics Array) при помощи FPGA.

Стандарт VGA был представлен в 1987 году для компьютеров IBM PS/2 с разрешением 640×480 пикселей на электронно-лучевой трубке (ЭЛТ) и 15-контактным разъемом, через который аналоговыми напряжениями передавалась информация о цвете. Современные ЖК-мониторы имеют более высокое разрешение, но сохраняют обратную совместимость со стандартом VGA.

В ЭЛТ электронная пушка сканирует экран слева направо, вызывая свечение флуоресцентного материала для отображения изображения. В цветных ЭЛТ используются три разных люминофора для красного, зеленого и синего цветов и три электронных луча. Мощность каждого луча определяет интенсивность соответствующего цвета в пикселе. В конце каждой строки развертки пушка должна выключиться на интервал горизонтального гашения, пока луч возвращается к началу следующей строки. После того как все растровые строки пройдены, пушку необходимо отключить снова на интервал вертикального гашения (обратного хода луча), чтобы луч вернулся в верхний левый угол. Процесс повторяется около 60–75 раз в секунду, чтобы создать визуальную иллюзию постоянного изображения. В современных дисплеях обычно используется ЖК-технология, которая не требует построчного сканирования, но для совместимости использует импульсы синхронизации интерфейса VGA.

В VGA-дисплее с разрешением 640×480 пикселей полный экран состоит из 800 пикселей × 525 горизонтальных строк развертки, как показано на **рис. 9.21**, но только 480 строк развертки и 640 пикселей в строке развертки фактически передают изображение, в то время как остальная часть остается черной. Сканирование линии начинается со *строчного гасящего импульса* (СГИ), так называемого «*заднего крыльца*» (back porch) – пустой области в левой части экрана. За ним следуют 640 пикселей, после чего идет *кадровый гасящий импульс* (КГИ), так называемое «*переднее крыльцо*» (front porch) в правой части экрана и *импульс горизонтальной синхронизации* (hsync), который возвращает луч к левому краю. В вертикальном направлении экран начинается с пустой области в 32 строки развертки вверху, за которой следуют 480 активных строк развертки, затем пустая область из 11 строк развертки внизу и импульс вертикальной синхронизации (vsync) с 2 строками развертки для возврата наверх, чтобы начать следующий кадр. Для монитора VGA с разрешением 640×480 пикселей, обновляемого с частотой 59,52 Гц, частота следования пикселей составляет 800×525×59,52 = 25 МГц, поэтому на каждый пиксель приходится 40 нс.

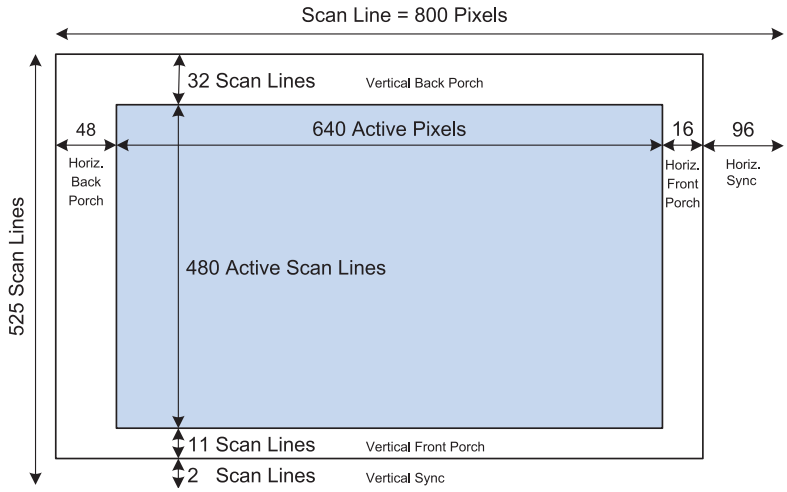
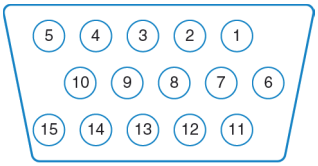


Рис. 9.21 Структура кадра VGA

На **рис. 9.22 (a)** показана временная диаграмма синхронизации строки растра. Полная длительность строки составляет 32 мкс. На **рис. 9.22 (b)** показана вертикальная синхронизация; обратите внимание, что теперь единицами времени являются строки развертки, а не пиксельные такты. Кадр обновляется приблизительно 60 раз в секунду. Более высокие разрешения используют и более высокую частоту тактирования пикселей, до 388 МГц для разрешения 2048×1536 при частоте кадров 85 Гц. Например, разрешение 1024×768 при частоте кадров 60 Гц может быть получено с частотой пиксельных тактов, равной 65 МГц.

Рис. 9.22 Временные диаграммы синхроимпульсов VGA: (a) по горизонтали, (b) по вертикали



1: Red
2: Green
3: Blue
4: Зарезервир.
5: GND
6: GND
7: GND
8: GND
9: 5 В (опционально)
10: GND
11: Зарезервировано
12: I^2C данные
13: HSync
14: Vsync
15: I^2C такты

Рис. 9.23 Схема расположения выводов разъема VGA

На **рис. 9.23** представлена схема расположения выводов разъема от источника видеосигнала. Информация о цвете пикселя передается тремя аналоговыми напряжениями в каналах красного, зеленого и синего цветов. Каждое напряжение находится в диапазоне от 0 до 0,7 В; чем выше напряжение, тем ярче цветовая составляющая. Напряжения должны быть равны нулю во время обратного хода луча. Видеосигнал необходимо генерировать с высокой скоростью в режиме реального времени, что сложно сделать с помощью микроконтроллера, но легко выполнимо на FPGA. Простое черно-белое изображение может быть получено путем подачи на все три входа цветовых компонентов напряжения 0 или 0,7 В с использованием делителя напряжения, подключенного к цифровому выходу схемы. Для формирования цветного изображения используют три независимых видео-ЦАП, с выходов которых аналоговые сигналы поступают на входы цветовых составляющих.

На **рис. 9.24** показана FPGA, формирующая изображение на VGA-мониторе при помощи тройного 8-битного видео-ЦАП ADV7125. Микросхема видео-ЦАП получает от FPGA 8-битные цифровые значения компонентов R, G и B. Она также получает сигнал SYNC_b, который принимает активный низкий уровень всякий раз, когда становится активным сигнал HSYNC или VSYNC. Токовые выходы видео-ЦАП управляют аналоговыми линиями красного, зеленого и синего цветов, которые обычно являются 75-омными параллельными линиями передачи, подключенными к видео-ЦАП и монитору. Резистор R_{SET} определяет величину выходного тока для достижения полного спектра цвета. Тактовая частота зависит от разрешения и частоты обновления. В быстродействующем ЦАП модели ADV7125JSTZ330 она может достигать значения 330 МГц.

Пример 9.11 ВЫВОД НА VGA-МОНИТОР

Используя схему, показанную на **рис. 9.24**, разработайте HDL-код для отображения текста и зеленого прямоугольника на VGA-мониторе.

Решение Мы предполагаем, что системная тактовая частота составляет 50 МГц, и используем делитель тактовой частоты для генерации тактовых импульсов VGA с частотой 25 МГц. Вы также можете использовать в качестве источника тактовых импульсов генератор с фазовой автоподстройкой частоты (ФАПЧ). Конфигурация ФАПЧ различается в зависимости от микросхемы FPGA; для Cyclone III частоты, генерируемые ФАПЧ, настраиваются с помощью специального мастера настройки от Altera (Intel FPGA). Как альтернативный вариант синхросигнал VGA может поступать от внешнего генератора сигналов.

Контроллер VGA считает столбцы и строки экрана, генерируя в нужное время сигналы `hsync` и `vsync`. Он также генерирует сигнал `blank_b`, который гасит луч, чтобы пространство за пределами активной области 640×480 было черным.

Генератор видео выдает значения красного, зеленого и синего цветов на основе текущего положения (x, y) пикселя на экране. Пиксель с координатами (0, 0) расположен в левом верхнем углу экрана. Генератор выводит на экран набор символов и зеленый прямоугольник. Изображения имеют размер 8×8 пикселей. Таким образом, рабочая область экрана вмещает 80×60 символов. Генератор получает изображение символа из ПЗУ, где оно представлено в виде битовой таблицы из 8 строк и 6 столбцов. Еще два столбца пустые, т. е. заполнены нулями. Код SystemVerilog меняет порядок битов на обратный, поскольку крайний левый столбец в файле ПЗУ является самым старшим битом, в то время как он должен отображаться в наименее значимой позиции x.

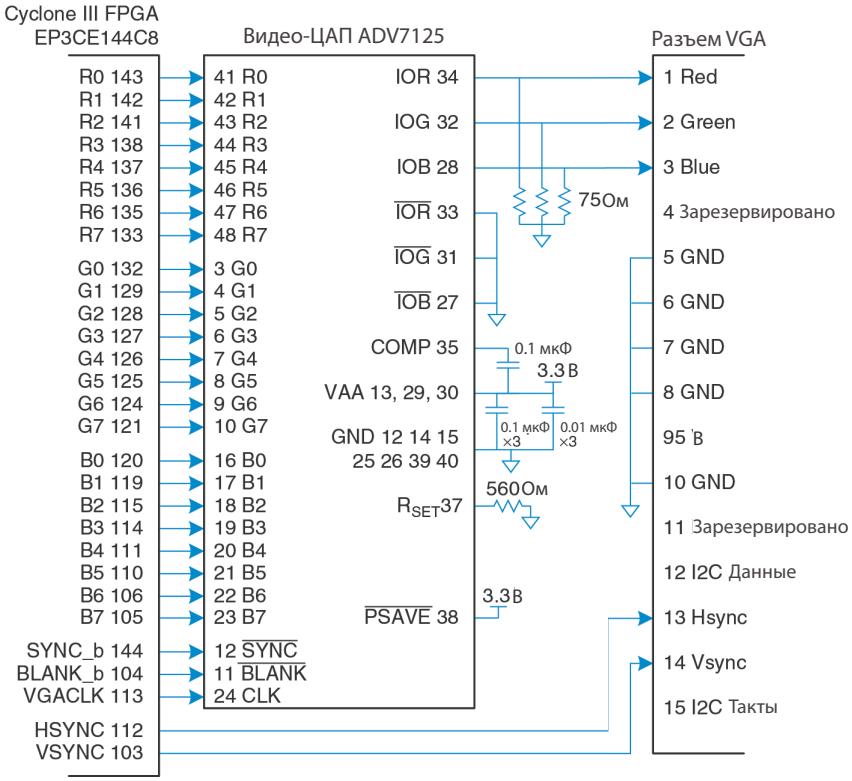


Рис. 9.24 Формирование видеосигнала при помощи FPGA и видео-ЦАП

На **рис. 9.25** показана фотография монитора VGA во время работы данной программы. Строки синего и красного цветов чередуются через одну. Зеленый прямоугольник перекрывает часть изображения.

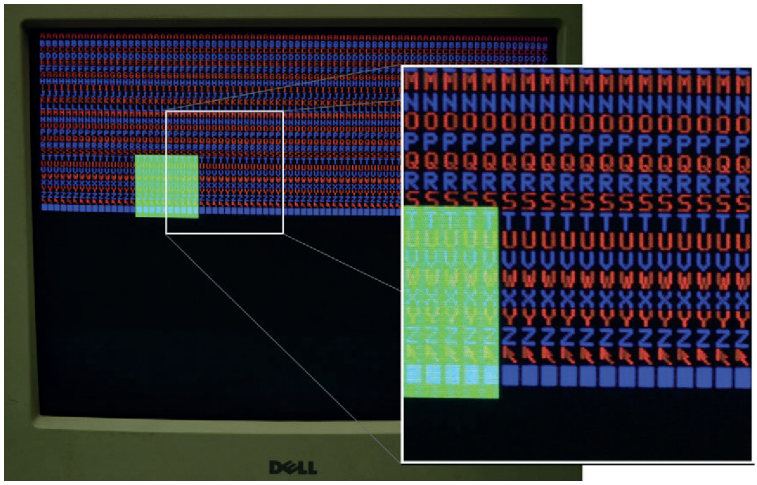


Рис. 9.25 Изображение на экране VGA-монитора

HDL-пример 9.3 КОД ИЗ ФАЙЛА vga.sv

```
module vga(input logic clk, reset,
           output logic vgaclk, // Тактовые импульсы VGA 25 МГц
           output logic hsync, vsync,
           output logic sync_b, blank_b, // выходы на монитор и ЦАП
           output logic [7:0] r, g, b);  // выходы на видео-ЦАП

  logic [9:0] x, y;

  // разделить входную частоту 50 МГц на 2, чтобы получить 25 МГц
  always_ff @(posedge clk, posedge reset)
    if (reset) vgaclk = 1'b0;
    else vgaclk = ~vgaclk;

  // генерация синхросигналов монитора
  vgaController vgaCont(vgaclk, reset, hsync, vsync, sync_b, blank_b, x, y);

  // определяемый пользователем модуль формирования цвета пикселя
  videoGen videoGen(x, y, r, g, b);

endmodule

module vgaController #(parameter HBP = 10'd48,       // горизонтальная темная область
                                 HACTIVE = 10'd640, // количество пикселей в строке
                                 HFP = 10'd16,       // горизонтальная темная область
                                 HSYN = 10'd96,      // строчный синхроимпульс = 60
                                                     // для возврата луча влево
                                 // количество пикселей по горизонтали (т. е. тактов на строку)
                                 HMAX = HBP + HACTIVE + HFP + HSYN, //48+640+16+96=800:
                                 VBP = 10'd32,       // вертикальная темная область
                                 VACTIVE = 10'd480, // количество строк
                                 VFP = 10'd11,       // вертикальная темная область
                                VSYN = 10'd2,       // кадровый синхроимпульс = 2
                                                     // для возврата луча вверх
                                // количество пикселей по вертикали (т. е. тактов)
                                 VMAX = VBP + VACTIVE + VFP + VSYN) //32+480+11+2=525:
```

HDL-пример 9.3 (окончание)

```
        (input logic vgaclk, reset,
         output logic hsync, vsync, sync_b, blank_b,
         output logic [9:0] hcnt, vcnt);

         // счетчики положения по вертикали и горизонтали
         always @(posedge vgaclk, posedge reset) begin
           if (reset) begin
             hcnt <= 0;
             vcnt <= 0;
           end
           else begin
             hcnt++;
               if (hcnt = = HMAX) begin
                hcnt <= 0;
                    vcnt++;
                    if (vcnt = = VMAX)
                      vcnt <= 0;
           end
       end
     end     // вычисление синхросигналов (активный низкий уровень)
     assign hsync = ~( (hcnt >= (HACTIVE + HFP)) & (hcnt < (HACTIVE + HFP + HSYN)) );
     assign vsync = ~( (vcnt >= (VACTIVE + VFP)) & (vcnt < (VACTIVE + VFP + VSYN)) );
     assign sync_b = 1'b0; // значение 0 для новых мониторов
                           // для старых мониторов: assign sync_b = hsync & vsync;
     // задать на выходах черный цвет вне рабочей области дисплея
     assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(input logic [9:0] x, y, output logic [7:0] r, g, b);
  logic pixel, inrect;

  // выбрать выводимый знак по координате y, затем выбрать
  // значение пикселя из ПЗУ знакогенератора и вывести его на экран
  // красным или синим. Потом нарисовать зеленый прямоугольник.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]= =0) ? {{8{pixel}},8'h00} : {8'h00, {8{pixel}}};
  assign g = inrect ? 8'hFF : 8'h00;
endmodule

module chargenrom(input logic [7:0] ch,
                  input logic [2:0] xoff, yoff,
                  output logic pixel);
  logic [5:0] charrom[2047:0]; // ПЗУ знакогенератора
  logic [7:0] line;            // чтение строки из ПЗУ

  // инициализировать ПЗУ знаками из текстового файла
  initial $readmemb("charrom.txt", charrom);

  // индекс в ПЗУ для поиска строки знаков
  assign line = charrom[yoff+{ch-65, 3'b000}]; // вычесть 65
                                               // поскольку A - нулевой адрес
  // обратить порядок битов
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input logic [9:0] x, y, left, top, right, bot,
               output logic inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

HDL-пример 9.4 СОДЕРЖИМОЕ ПЗУ СИМВОЛОВ (charrom.txt)

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

9.4.3. Беспроводная связь Bluetooth

В настоящее время существует много стандартов беспроводной связи, в том числе Wi-Fi, ZigBee и Bluetooth. Эти стандарты детально проработаны и требуют использования сложных интегральных схем, но растущий ассортимент модулей позволяет абстрагироваться от сложности и предоставить пользователю простой интерфейс для беспроводной связи. Одним из этих модулей является BlueSMiRF, простой беспроводной адаптер Bluetooth, который можно использовать вместо последовательной передачи данных по кабелю.

Bluetooth – это беспроводной интерфейс, разработанный компанией Ericsson в 1994 году для маломощной связи на умеренной скорости на расстояниях 5–100 м, в зависимости от уровня мощности передатчика. Он широко используется для подключения гарнитуры к телефону или клавиатуры к компьютеру. В отличие от инфракрасных каналов связи, он не требует прямой видимости между устройствами.

Bluetooth работает в нелицензируемом диапазоне 2,4 ГГц для промышленного, научного и медицинского применения (ISM). Он определяет 79 радиоканалов, разнесенных с интервалом в 1 МГц, начиная с 2402 МГц. Он переключается между этими каналами в псевдослучайной последовательности, чтобы не создавать постоянные помехи дру-

гим устройствам, например беспроводным маршрутизаторам, работающим в том же диапазоне. Как указано в **табл. 9.13**, передатчики Bluetooth классифицируются по одному из трех уровней мощности, которые определяют их дальность и энергопотребление. В базовом режиме он работает со скоростью 1 Мбит/с, используя частотную манипуляцию (frequency-shift keying, FSK). В обычной FSK каждый бит передается излучением с частотой $f_c \pm f_d$, где f_c – центральная частота канала, а f_d – смещение частоты не менее 115 кГц. Резкий переход частот между битами занимает дополнительную полосу пропускания. В гауссовой FSK переходы между частотами сглажены, что позволяет более эффективно использовать спектр частот. На **рис. 9.26** показано чередование частот при передаче последовательности нулей и единиц на канале 2402 МГц с использованием FSK и GFSK (Gaussian FSK).

Харальд Синезубый

(*По правде говоря, это шведский гитарист Олов Киндгрен, но мы полагаем, что король Харальд Синезубый выглядел примерно так же. Фотография напечатана с разрешения автора снимка.*) Стандарт Bluetooth (синий зуб) назван так в честь короля Дании Харальда Синезубого, монарха X века, который объединил враждующие датские племена. Правда, этот беспроводной стандарт добился лишь частичного успеха в деле унификации конкурирующих протоколов беспроводной связи!

Таблица 9.13 Классы Bluetooth

Класс	Мощность передатчика, мВт	Дальность, м
1	100	100
2	2,5	10
3	1	5

Модуль BlueSMiRF Silver, приведенный на **рис. 9.27 (a)**, содержит радиомодуль Bluetooth класса 2, модем и схему последовательного интерфейса на небольшой плате. Он обменивается данными с другим устройством Bluetooth, например ноутбуком со встраиваемым модулем Bluetooth, или USB-адаптером Bluetooth, подключенным к ПК. Используя модуль Bluetooth, можно организовать беспроводное последовательное соединение между RED-V и ПК, аналогичное соединению на **рис. 9.13**, но без кабеля. Беспроводная связь совместима с тем же программным обеспечением, что и проводная связь.

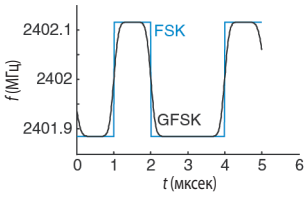


Рис. 9.26 Временные диаграммы сигналов FSK и GFSK

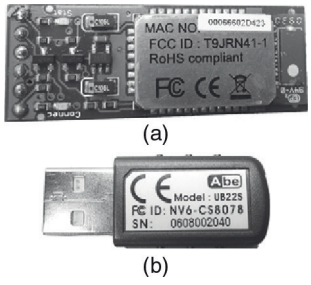


Рис. 9.27 Модуль BlueSMiRF (a) и USB-адаптер (b)

На **рис. 9.28** показана схема такого соединения. Вывод TX модуля BlueSMiRF подключается к выводу RX платы RED-V, и наоборот. Контакты RTS и CTS соединены перемычкой, так что BlueSMiRF выполняет хендшейк (запрос-ответ установления связи) с самим собой.

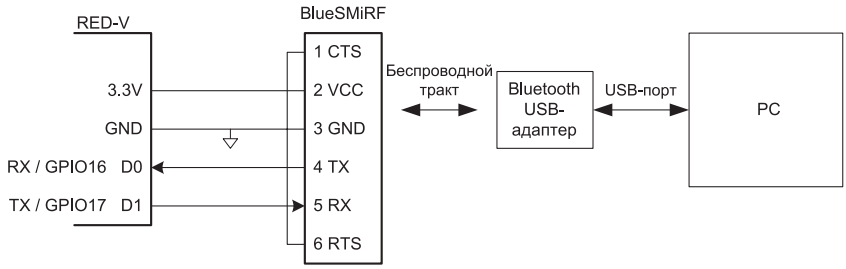


Рис. 9.28 Схема подключения RED-V к ПК при помощи BlueSMiRF

Модуль BlueSMiRF по умолчанию работает на скорости 115,2 Кбод с 8 битами данных, 1 стоповым битом и без контроля четности или управления потоком. Он работает с логическими уровнями 3,3 В, поэтому его можно напрямую соединять с устройствами, использующими цифровую логику с уровнями 3,3 В.

Чтобы использовать этот интерфейс, подключите USB-адаптер Bluetooth к ПК. Включите RED-V и BlueSMiRF. Красный индикатор STAT на BlueSMiRF будет мигать, указывая на то, что он ожидает установки соединения. Нажмите на значок Bluetooth на панели задач ПК и используйте мастер добавления устройства Bluetooth для сопряжения USB-адаптера с модулем BlueSMiRF. Ключ доступа по умолчанию для BlueSMiRF – 1234. Обратите внимание, какой виртуальный COM-порт назначен USB-адаптеру. Через этот порт связь может работать так же, как по последовательному кабелю. Обратите внимание, что по умолчанию адаптер работает на скорости 9600 бод и что PuTTY должен быть настроен соответствующим образом.

9.4.4. Управление двигателями

Еще одним важным применением микроконтроллеров является управление исполнительными механизмами, такими как двигатели. В этом разделе описываются три типа двигателей: двигатели постоянного тока, серводвигатели и шаговые двигатели. *Двигатели постоянного тока* потребляют большой ток – обычно около 1 А, – поэтому между микроконтроллером и двигателем следует включить мощную схему управления, такую как *H-мост*. Они также нуждаются в отдельном датчике угла поворота, если пользователь хочет знать текущее положение вала двигателя. *Серводвигатели* принимают ШИМ-сигнал, чтобы обозначить свое положение в ограниченном диапазоне углов. С ними очень легко взаи-

модействовать, но они не такие мощные и не подходят для продолжительного непрерывного вращения. *Шаговые двигатели* принимают последовательность импульсов, каждый из которых вращает двигатель на фиксированный угол, называемый шагом. Они стоят дороже, и им тоже нужен H-мост для управления большим током, но положение подвижной части двигателя можно точно контролировать.

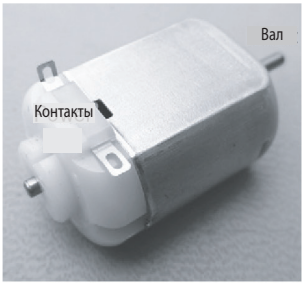
Двигатели могут потреблять значительный ток, что может привести к сбоям в работе цифровой логики из-за помех по цепям питания. Один из способов устранения этой проблемы заключается в использовании одного источника питания или батареи для питания двигателя и другого источника питания – для цифровой логики.

Электродвигатели постоянного тока

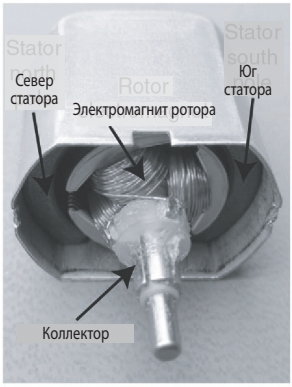
На **рис. 9.29** показана конструкция щеточного двигателя постоянного тока. Такой двигатель представляет собой устройство с двумя выводами. Он содержит постоянные неподвижные магниты, называемые статорами, и вращающийся электромагнит, называемый ротором, или якорем, подключенный к валу. Передний конец ротора соединяется с металлическим кольцом, называемым *коллектором* (commutator). Металлические щетки, присоединенные к выводам питания, скользят по коллектору, обеспечивая протекание тока через электромагнит ротора. Электрический ток индуцирует магнитное поле в роторе, заставляющее ротор вращаться в магнитном поле статора. После того как ротор проходит часть пути при вращении и приближается к выравниванию со статором, щетки касаются противоположных контактов коллектора, изменяя направление тока и магнитного поля на противоположное и заставляя его непрерывно вращаться.

Двигатели постоянного тока, как правило, вращаются, совершая тысячи оборотов в минуту (rotations per minute, RPM) при очень низком крутящем моменте. В большинство систем добавляют редуктор, чтобы уменьшить скорость вращения до более приемлемого уровня и увеличить крутящий момент. Постарайтесь подобрать редуктор, подходящий к вашему двигателю. Компания Pittman производит широкий спектр высококачественных двигателей постоянного тока и аксессуаров к ним, в то время как недорогие игрушечные двигатели популярны среди любителей.

Чтобы передать на нагрузку высокую механическую мощность, двигатель должен потреблять значительный электрический ток. Направление тока должно быть обратимым, если



(a)



(b)

(c)

Рис. 9.29 Двигатель постоянного тока

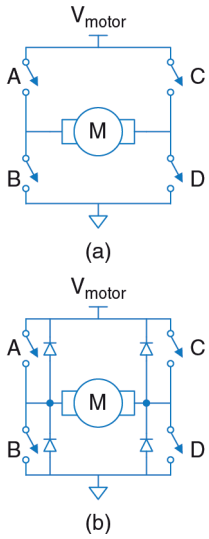


Рис. 9.30 **Н-мост**

необходимо, чтобы двигатель вращался в обоих направлениях. Большинство микроконтроллеров не могут коммутировать достаточный ток для управления двигателем постоянного тока напрямую. Вместо этого они используют *Н-мост*, который конструктивно содержит четыре электрически управляемых ключа, как показано на **рис. 9.30 (a)**. Он получил такое название, потому что схема коммутации похожа на букву Н. Если ключи A и D замкнуты, ток течет слева направо, через двигатель, и он вращается в одном направлении. Если ключи B и C замкнуты, ток течет справа налево через двигатель, и он вращается в другом направлении. Если одновременно замкнуты ключи A и C или B и D, напряжение на двигателе равно нулю, в результате чего двигатель активно тормозится. Если разомкнуть все ключи, то двигатель будет вращаться по инерции до полной остановки. Ключи в Н-мосте представляют собой силовые транзисторы, способные выдерживать высокие токи в один или несколько ампер. Н-мост также содержит несложную цифровую логику для удобного управления ключами. Микроконтроллер подключают к слаботочным цифровым входам Н-моста для управления силовыми ключами.

При резком изменении тока через двигатель индуктивность обмоток вызовет большой скачок напряжения, который может повредить силовые транзисторы. Поэтому многие Н-мосты также имеют *защитные диоды*, подключенные параллельно ключам, как показано на **рис. 9.30 (b)**. Если напряжение индуктивности импульса оказывается выше питающего напряжения V_{motor} или ниже уровня GND, диоды открываются и ограничивают напряжение на безопасном уровне. Н-мосты могут рассеивать большое количество энергии, поэтому для их охлаждения может потребоваться схема отвода тепла.

Пример 9.12 АВТОНОМНЫЙ АВТОМОБИЛЬ

Спроектируйте систему, в которой плата RED-V управляет двумя приводными электродвигателями автомобиля-робота. Разработайте библиотеку, состоящую из функций инициализации драйвера[1] двигателя и управления машиной, чтобы она двигалась вперед и назад, поворачивала вправо, влево и останавливалась. Используйте ШИМ для изменения выходного напряжения и, таким образом, управления скоростью вращения двигателей.

Решение На **рис. 9.31** приведена схема, где два электродвигателя постоянного тока управляются платой RED-V с использованием микросхемы Texas Instruments SN754410 (двойной Н-мост). Этой микросхеме требуется питающее напряжение 5 В, поступающее на V_{CC1}, а также напряжение 4,5–36 В для питания двигателей, поступающее на V_{CC2}. Напряжение V_{IH} на входах драйвера со-

[1] Микросхемы, которые содержат Н-мост или иную схему управления электродвигателями, обычно называют драйверами двигателя. – *Прим. перев.*

ставляет 2 В, то есть этот порт совместим с портами ввода / вывода платы RED-V, которые работают с логическими уровнями 3,3 В. Драйвер может коммутировать ток до 1 А через каждый из двух двигателей. Напряжение V_{motor} должно поступать от отдельного источника питания (аккумулятора); выход 5 В платы RED-V не может обеспечить достаточный ток для управления большинством двигателей, и поэтому плата может выйти из строя.

Таблица 9.14 показывает, как каждый из двух H-мостов управляет вращением двигателей. Микроконтроллер управляет скоростью вращения, используя ШИМ. Для управления направлением вращения роторов двигателей используются четыре других входа драйвера.

Таблица 9.14
Управление H-мостом

EN12	1A	2A	Мотор
0	X	X	X
1	0	0	Тормоз
1	0	1	Реверс
1	1	0	Вперед
1	1	1	Тормоз

ШИМ настроен на работу на частоте около 5 кГц с коэффициентом заполнения от 0 % до 100 %. Любая частота ШИМ, намного превышающая полосу пропускания двигателя, даст эффект плавного движения. Обратите внимание, что соотношение между коэффициентом заполнения и скоростью вращения двигателя является нелинейным и что ниже некоторого значения заполнения ШИМ вал двигателя просто не будет вращаться.

В **примере кода 9.18** показано, как использовать ШИМ-управление в схеме с двойным H-мостом (**рис. 9.31**) для управления двумя двигателями постоянного тока.

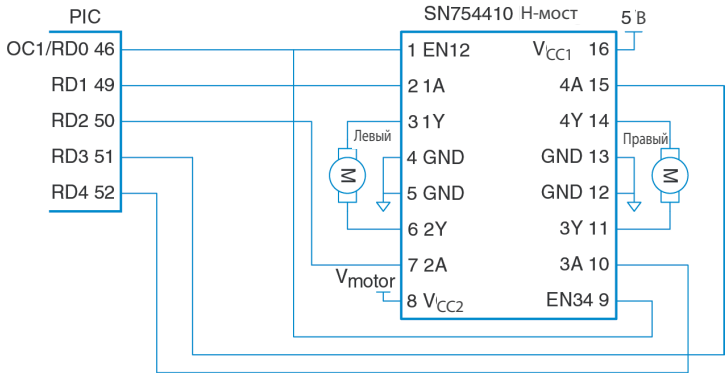


Рис. 9.31 Управление двигателями при помощи двойного H-моста

Пример кода 9.18 ДРАЙВЕР ДВИГАТЕЛЯ ПОСТОЯННОГО ТОКА

```
#include "EasyREDVIO.h"

// Константы, используемые для подключения двигателей
#define EN D3
#define MOTOR_1A D4
#define MOTOR_2A D5
#define MOTOR_3A D6
#define MOTOR_4A D7
```

Пример кода 9.18 (окончание)

```
void setMotorLeft(int dir) {  // направление 1 = вперед, 0 = назад
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir) { // направление 1 = вперед, 0 = назад
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1); // оба мотора вперед
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0); // оба мотора назад
}

void left(void) {
    setMotorLeft(0); setMotorRight(1); // левый назад, правый вперед
}

void right(void) {
    setMotorLeft(1); setMotorRight(0); // правый назад, левый вперед
}

void halt(void) { // оба мотора выключены
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                       // убедимся, что моторы остановлены
    pwmInit(EN, 1, 255);          // включаем ШИМ
    analogWrite(200);             // по умолчанию подается не вся мощность
}

int main(void) {
    initMotors();
    while(1)
    {
        forward();
        delay(5000);
        backward();
        delay(5000);
        left();
        delay(5000);
        right();
        delay(5000);
        halt();
    }
}
```

В предыдущем примере нет способа измерить положение каждого двигателя. Два двигателя вряд ли будут в точности повторять состояние друг друга, так что, скорее всего, один будет вращаться немного быстрее другого, в результате чего робот отклонится от курса. Чтобы решить

эту проблему, в некоторых системах добавляют *датчики угла поворота* (датчики углового положения, энкодеры). На **рис. 9.32 (a)** показан простой оптический энкодер, состоящий из диска с прорезями, прикрепленного к валу двигателя. С одной стороны диска размещен светодиод, с другой – датчик освещенности (Оптическая пара. – *Прим. перев.*). Энкодер выдает импульс каждый раз, когда прорезь проходит через оптическую пару. Микроконтроллер может подсчитывать эти импульсы, чтобы измерить общий угол поворота вала. При помощи двух оптических пар, разнесенных на половину ширины прорези, более совершенный энкодер может выдавать квадратурные выходные сигналы, показанные на **рис. 9.32 (b)**, указывающие не только на угол поворота вала, но и направление вращения. Иногда в диске энкодера делают еще одно отверстие, которое указывает на начальное положение вала.

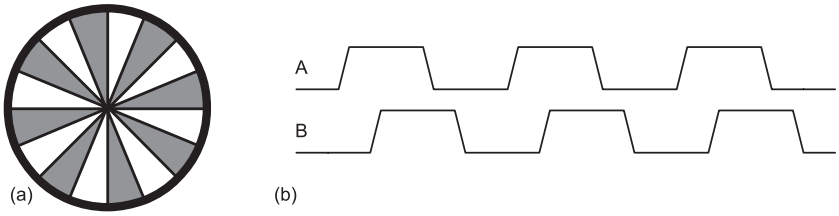


Рис. 9.32 Оптический энкодер: (a) диск, (b) квадратурные выходные сигналы

Серводвигатель

Серводвигатель – это двигатель постоянного тока, интегрированный с зубчатой передачей, датчиком углового положения вала и некоторой управляющей логикой, поэтому он проще в использовании. Такие двигатели имеют ограниченный угол поворота, обычно 180°. На **рис. 9.33** изображен серводвигатель со снятой крышкой, чтобы показать внутренний механизм. Серводвигатель имеет 3-контактный интерфейс с выводами питания (обычно 5 В) и земли, а также управляющим входом. На управляющий вход, как правило, подают ШИМ-сигнал с рабочей частотой 50 Гц. Логика управления серводвигателем приводит вал в положение, определяемое коэффициентом заполнения входного сигнала управления. В качестве энкодера серводвигателя, как правило, выступает поворотный потенциометр, который создает напряжение, прямо пропорциональное углу поворота вала.

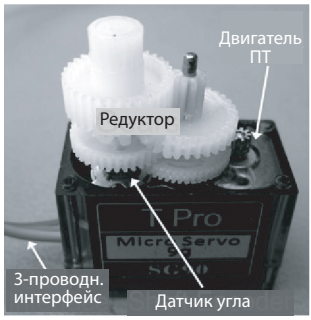


Рис. 9.33 Серводвигатель SG90

В типичном серводвигателе с вращением вала до 180° подача импульса длительностью 1 мс устанавливает вал в положение 0°, 1,5 мс вызывает поворот на 90°, а от 2,5 мс – на 180°. Например, на **рис. 9.34** показан управляющий сигнал с длительностью импульса 1,5 мс. Выход сервопри-

вода за пределы допустимого диапазона может привести к столкновению вала с механическими ограничителями и поломке сервопривода. Питание серводвигателя приходит от вывода питания, а не вывода управления, поэтому управление можно подключить непосредственно к микроконтроллеру без H-моста. Серводвигатели обычно используются в моделях самолетов с дистанционным управлением и маленьких роботах из-за их размеров, легкости и удобства. Обычно довольно трудно найти серводвигатель с подробной технической документацией. Центральный контакт с красным проводом, как правило, является выводом питания, а черный или коричневый провод – это обычно земля.

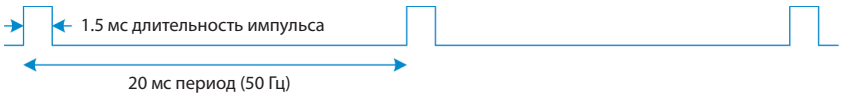


Рис. 9.34 Временная диаграмма управляющего сигнала серводвигателя

Пример 9.13 СЕРВОДВИГАТЕЛЬ

Разработайте систему управления на основе платы RED-V, которая поворачивает вал серводвигателя на заданный угол.

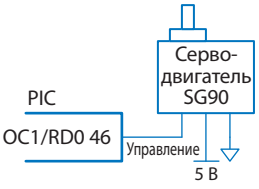


Рис. 9.35 Управление серводвигателем

Решение На **рис. 9.35** показана схема подключения серводвигателя SG90, включая цвет соединительных проводов. Серводвигатель питается от источника постоянного тока с напряжением от 4,0 до 7,2 В. Серводвигатель SG90 может потреблять ток до 0,5 А, если вынужден развивать значительное усилие на валу, но при небольших механических нагрузках его можно запитать от того же источника, что и плату RED-V. Для передачи сигнала от ШИМ достаточно одного провода, по которому передается импульсный сигнал с напряжением 5 В или 3,3 В. Программа из **примера кода 9.19** выполняет настройку генератора ШИМ и вычисляет коэффициент заполнения импульсов, который необходим для поворота вала двигателя на заданный угол. Программа циклически устанавливает вал в положение 0°, 90° и 180°.

Обычный серводвигатель можно превратить в серводвигатель непрерывного вращения, осторожно разобрав его, удалив механический упор и заменив потенциометр фиксированным делителем напряжения. На многих веб-сайтах приведены подробные инструкции для конкретных серводвигателей. В таком случае ШИМ будет контролировать скорость, а не положение: длительность импульса 1,5 мс означает остановку, 2,5 мс – вращение на полной скорости в прямом направлении и 0,5 мс – вращение на полной скорости в обратном направлении. Непрерывно вращающийся серводвигатель может быть более удобным и менее дорогим, чем простой двигатель постоянного тока в сочетании с H-мостом и зубчатой передачей.

Пример кода 9.19 УПРАВЛЕНИЕ СЕРВОДВИГАТЕЛЕМ

```
#include "EasyREDVIO.h"

void genPulseMicroseconds(uint16_t pulse_len_us) {
    PWM1->pwmcmp1.pwmcmp = pulse_len_us;
}

void setServo(float angle) {
    volatile uint16_t pulse_len_us = (uint16_t) (1000 + (angle / 180) * 1000);
    genPulseMicroseconds(pulse_len_us);
}

int main(void) {
    uint32_t scale = 4; // Настройка делителя для получения частоты 16e6/2^4 = 1 МГц,
                        // которая нужна для достижения точности 1 мкс
    float freq = 50.0;
    volatile uint32_t pwm_period_count = (uint32_t) (1/freq * 1e6); // Период ШИМ
                                                                    // в микросекундах
    pwmInit(D3, scale, pwm_period_count);
    while(1) {
        setServo(0.0);
        delay(1000);
        setServo(90.0);
        delay(1000);
        setServo(180.0);
        delay(1000);
    }
}
```

Шаговый двигатель

Вал шагового двигателя поворачивается на дискретный угол (шаг), по мере того как на его входы поочередно подаются электрические импульсы. Шаг обычно составляет несколько градусов, что позволяет выполнить точное позиционирование и продолжительное вращение. Малые шаговые двигатели, как правило, имеют два набора катушек, называемых фазами, исполненных в биполярном или однополярном виде. Биполярные двигатели мощнее и дешевле при заданном размере, но требуют использования H-моста в качестве драйвера, в то время как униполярные двигатели могут управляться транзисторами, работающими как переключатели. В этом разделе рассматривается более эффективный биполярный шаговый двигатель.

На **рис. 9.36 (a)** показан упрощенный двухфазный биполярный двигатель с шагом 90°. Ротор является постоянным магнитом с одним северным и одним южным полюсами. Статор – электромагнит с двумя парами катушек, содержащий две фазы. Двухфазные биполярные двигатели, таким образом, имеют четыре терминала. На **рис. 9.36 (b)** показана

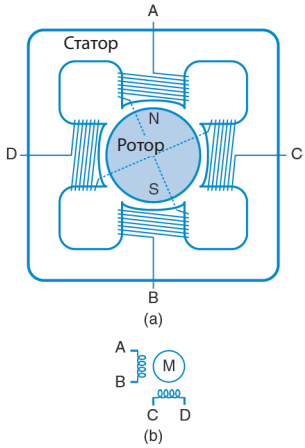


Рис. 9.36 a) Упрощенный биполярный шаговый двигатель, b) символ шагового двигателя

схема шагового двигателя, состоящего из двух катушек индуктивности. На практике к двигателю добавляют зубчатый редуктор, чтобы уменьшить шаг вращения и увеличить крутящий момент.

На **рис. 9.37** показаны три общие управляющие последовательности для биполярного двухфазного двигателя. На **рис. 9.37 (а)** проиллюстрирован *привод вала*, в котором на катушки подается напряжение в последовательности AB – CD – BA – DC. Следует отметить, что BA означа-

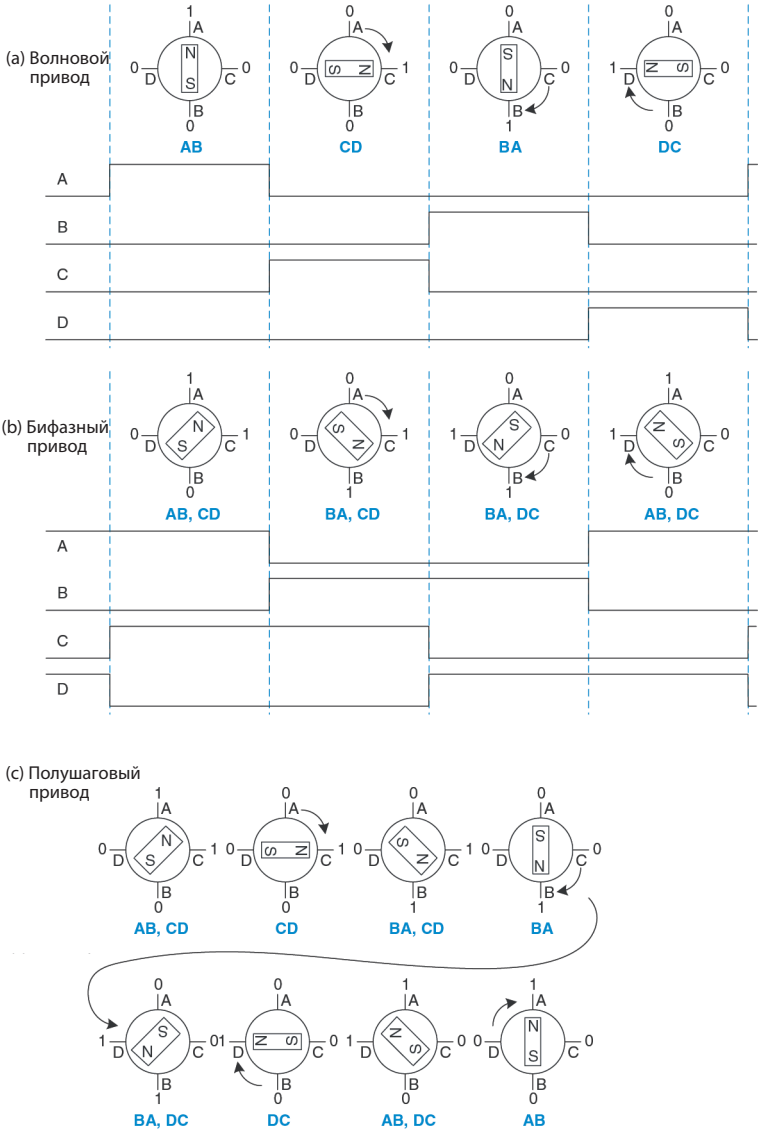


Рис. 9.37 Управление биполярным шаговым двигателем

ет, что обмотка AB находится под напряжением с током, направленным в обратную сторону; отсюда и возникло название «*биполярный*». Ротор поворачивается на 90° на каждом шаге. **Рисунок 9.37 (b)** иллюстрирует работу *привода с двумя одновременно включаемыми фазами*, работающего по последовательности (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) указывает на то, что обе катушки AB и CD находятся под напряжением одновременно. В этом случае ротор тоже поворачивается на 90° на каждом шаге, но производит автоматическое выравнивание на полпути между двумя полюсами. Это дает самое быстрое вращение, так как обе катушки работают одновременно. На **рис. 9.37 (c)** показан привод с *полушагом*, работающий по схеме (AB, CD) – CD (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. Ротор вращается на 45 градусов в каждом полушаге. Скорость выполнения этой последовательности определяет скорость вращения двигателя. Чтобы изменить направление вращения двигателя, те же управляющие последовательности подаются в обратном порядке.

В реальном двигателе ротор имеет множество полюсов, чтобы угол между шагами был намного меньше. Например, на **рис. 9.39** показан биполярный шаговый двигатель AIRPAX LB82773-M1 с размером шага 7,5°. Двигатель работает от 5 В и потребляет 0,8 А на каждую катушку.

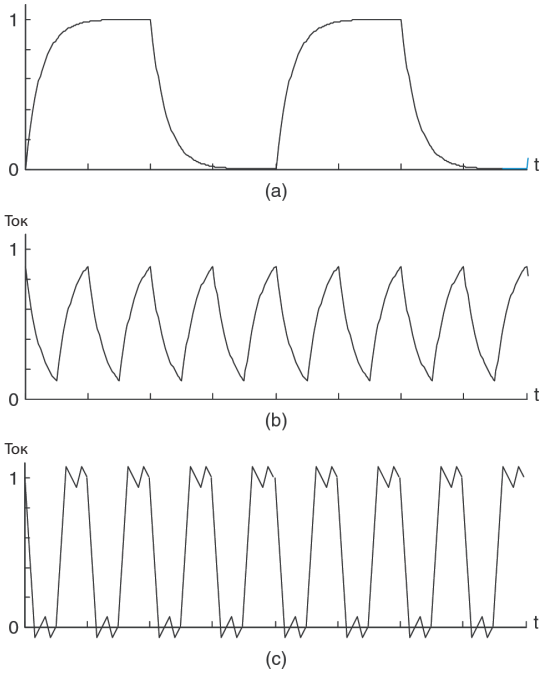


Рис. 9.38 Прямой ток в биполярном шаговом двигателе при: a) медленном вращении, b) быстром вращении, c) быстром вращении при ограничении постоянного тока

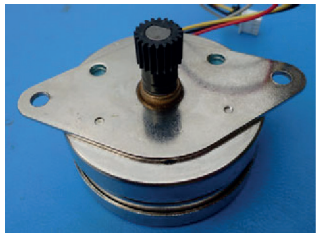


Рис. 9.39 Биполярный шаговый двигатель AIRPAX LB82773-M1

Крутящий момент двигателя пропорционален току, протекающему через катушку. Этот ток определяется приложенным напряжением, а также индуктивностью L и сопротивлением R катушки. Самый простой режим работы называют приводом постоянного напряжения, или L/R-приводом, в котором напряжение V прикладывается непосредственно к катушке. Ток достигает значения $I = V/R$ с постоянной времени, определяемой соотношением L/R, как показано на **рис. 9.38 (a)**. Это хорошо работает при малых скоростях. Но при более высокой скорости ток не успевает достичь максимального уровня, как показано на **рис. 9.38 (b)**, и крутящий момент снижается.

Более эффективный способ управления шаговым двигателем – метод широтно-импульсной модуляции более высокого напряжения. Высокое напряжение заставляет ток нарастать до максимального значения быстрее, после чего он выключается ШИМ, чтобы избежать перегрузки двигателя. Затем напряжение модулируется или ограничивается, чтобы поддерживать ток около нужного уровня. Шаговый двигатель с таким методом управления (как показано на **рис. 9.38 (c)**) называется приводом с ограничителем по постоянному току. Контроллер использует низкоомный резистор, включенный последовательно с двигателем, чтобы определить протекающий ток путем измерения падения напряжения, и посылает разрешающий сигнал H-мосту, чтобы выключить привод, когда ток достигает желаемого уровня. В принципе, микроконтроллер может генерировать сигналы нужной формы, но это проще сделать с помощью контроллера шагового двигателя. Контроллер L297 от ST Microelectronics – подходящий выбор, особенно в сочетании с двойным H-мостом L298 с выводами измерения тока и максимально допустимым пиковым током 2 А. К сожалению, L298 не доступен в DIP-корпусе, поэтому его сложнее монтировать на печатную плату. Документация компании ST на AN460 и AN470 очень полезна для разработчиков шаговых двигателей.

Пример 9.14 БИПОЛЯРНЫЙ ШАГОВЫЙ ДВИГАТЕЛЬ ПРЯМОГО ПРИВОДА

Разработайте систему для управления биполярным шаговым двигателем AIRPAX с заданной скоростью и направлением вращения, используя одновременно включаемые фазы.

Решение На **рис. 9.40** показан биполярный шаговый двигатель, управляемый непосредственно H-мостом с тем же интерфейсом, что и двигатель постоянного тока. Обратите внимание, что источник питания VCC2 должен обеспечивать достаточное напряжение и ток для удовлетворения требований двигателя, иначе двигатель может пропускать шаги при увеличении скорости вращения.

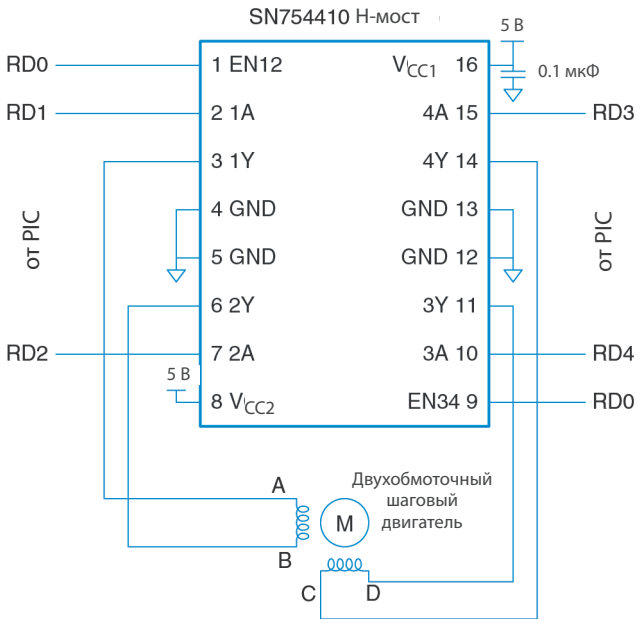


Рис. 9.40 **Биполярный шаговый двигатель, управляемый с помощью H-моста**

Пример кода 9.20 УПРАВЛЕНИЕ ШАГОВЫМ ДВИГАТЕЛЕМ

```
#include "EasyREDVIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MILLIS_PER_SEC 1000
#define DEG_PER_REV 360

int stepperPins[] = {19, 22, 23, 20, 21};
int curStepState; // храним текущее состояние шагового двигателя

void stepperInit(void) {
  pinsMode(stepperPins, 5, OUTPUT);
  curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001}; //{2A, 1A, 4A, 3A, EN}
  int step = 0;

  unsigned int millisPerStep = (SECS_PER_MIN * MILLIS_PER_SEC * STEPSIZE) /
                               (rpm * DEG_PER_REV);

  for (step = 0; step < steps; step++) {
    digitalWrites(stepperPins, 5, sequence[curStepState]);
```

Пример кода 9.20 (окончание)

```
    if (dir = = 0) curStepState = (curStepState + 1) % 4;
    else curStepState = (curStepState + 3) % 4;
    delay(millisPerStep);
  }
}

int main(void) {
  stepperInit();
  stepperSpin(1, 12000, 120); // Выполнить 60 шагов на скорости 120 об/мин
}
```

9.5. Заключение

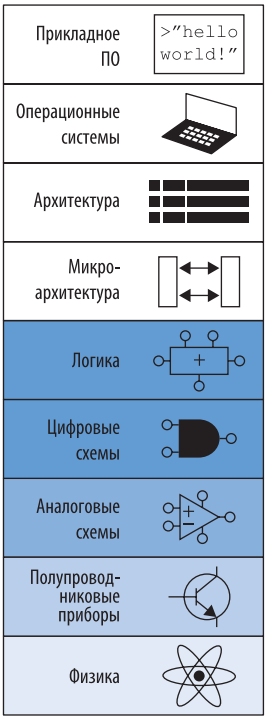
Большинство процессоров используют ввод/вывод с отображением в адресное пространство памяти для связи с окружающим миром. Для этого микроконтроллеры оснащают такими базовыми периферийными модулями, как универсальные, последовательные и аналоговые устройства ввода/вывода и таймеры.

В этой главе мы рассмотрели ряд конкретных примеров организации ввода/вывода с использованием микроконтроллера FE310 RISC-V на SparkFun RED-V RedBoard. Разработчики встраиваемых систем постоянно сталкиваются с новыми процессорами и периферийными устройствами. Общий принцип реализации функций ввода/вывода встраиваемой системы прост – нужно изучить карты памяти и регистров, чтобы определить, какие периферийные устройства доступны и какие выводы устройства и отображаемые в память регистры ввода/вывода задействованы в схеме. Затем, как правило, достаточно разработать простой драйвер устройства, который инициализирует регистры периферийного устройства и передает или принимает данные.

Для более сложных протоколов ввода/вывода, таких как USB, разработка драйвера устройства – это задача, требующая специальных знаний, с которой лучше всего справится специалист, хорошо знающий устройство и стек протоколов USB. Обычным разработчикам следует выбирать для своего устройства процессор с проверенными драйверами и примерами кода.

Приложение А

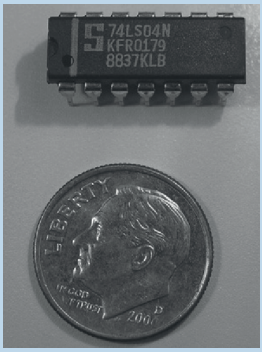
Реализация цифровых систем



А.1. Введение

В приложении описываются практические вопросы разработки цифровых систем. Этот материал не является необходимым для понимания содержания остальной части книги, но он проливает свет на процесс практической разработки цифровых систем. Более того, можно быть уверенным, что наилучший путь к пониманию цифровых систем – это их самостоятельная сборка и отладка в лаборатории.

Обычно цифровые системы состоят из одной или нескольких микросхем. Одна из технологий создания схем – соединение друг с другом микросхем, состоящих из отдельных логических вентилей или более крупных элементов – арифметико-логических устройств (АЛУ) или памяти. Другая технология – использование программируемой логики, которая состоит из набора стандартных элементов и может быть запрограммирована для выполнения требуемых логических операций. Третья техноло-



Микросхема инвертора 74LS04 представлена в 14-выводном корпусе DIP. Условное обозначение микросхемы находится в первой строке. Буквы LS обозначают семейство логических схем (раздел А.6). Суффикс N указывает на тип корпуса – DIP. S – это товарный знак производителя (Signetics). Комбинация цифр и букв в двух последних строчках – информация о заводской партии, в которой была изготовлена микросхема.

гия – разработка непосредственно для заказчика специализированной микросхемы, содержащей определенные логические элементы, необходимые для системы. Эти три технологии предлагают компромиссы по стоимости, быстродействию, энергопотреблению, времени разработки, которые рассмотрены ниже. В приложении также описаны корпуса и монтаж микросхем, линии передачи, соединяющие микросхемы и экономические аспекты разработки и изготовления цифровых систем.

А.2. Логические микросхемы серии 74xx

В 1970–1980-х годах многие цифровые системы строились из простых микросхем, содержащих лишь несколько логических элементов в чипе. Например, микросхема 7404 содержит шесть элементов НЕ, микросхема 7408 – четыре элемента И, микросхема 7474 – два триггера. Эти микросхемы относятся к серии логических элементов 74xx. Их продавали многие производители, обычно по цене от 10 до 25 центов за микросхему. По большей части эти микросхемы вышли из употребления, но они до сих пор полезны для создания простых цифровых систем и выполнения лабораторных работ, поскольку они дешевы и просты в использовании. Микросхемы серии 74xx обычно продаются в 14-выводных корпусах DIP (Dual In-line Package – корпуса с двухрядным расположением выводов).

А.2.1. Логические элементы

На **рис. А.1** показано расположение выводов для нескольких известных микросхем серии 74xx, содержащих основные логические элементы. Их иногда называют *малыми интегральными схемами* (МИС), поскольку они состоят всего лишь из нескольких транзисторов. 14-выводные корпуса обычно имеют вырез сверху или точку в левом верхнем углу для определения ориентации. Выводы нумеруются, начиная с 1 в верхнем левом углу и далее вокруг корпуса против часовой стрелки. Выводы 14 и 7 микросхемы следует подключить к питанию (V_{DD} = 5 В) и общему проводу (GND = 0 В) соответственно. Количество логических элементов в микросхеме определяется количеством выводов. Обратите внимание, что выводы 3 и 11 микросхемы 7421 ни к чему не подсоединены (*NC*). Триггер 7474 содержит стандартные выводы *D*, *CLK* и *Q*. Также у него есть инверсный выход $\overline{Q}$. Более того, у этого триггера есть входы асинхронной установки (также называемой предустановкой *PRE*) и сброса (*CLR*). Эти выводы активируются низким уровнем, другими словами,

триггер устанавливается, когда $\overline{PRE} = 0$, сбрасывается, когда $\overline{CLR} = 0$, и работает в нормальном режиме, когда $\overline{PRE} = \overline{CLR} = 1$.

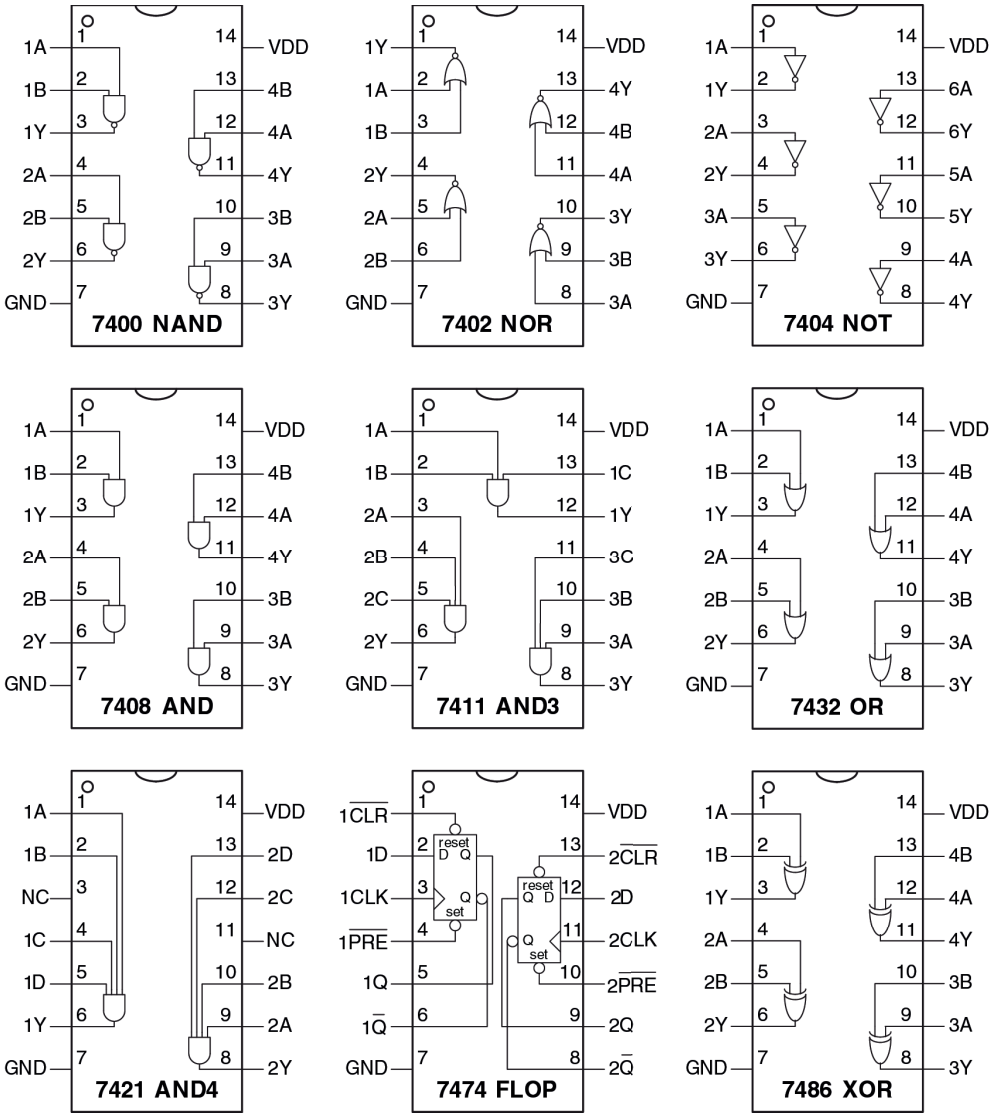
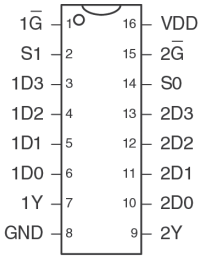


Рис. А.1 **Стандартные логические микросхемы серии 74xx**

A.2.2. Другие логические функции

Другие микросхемы серии 74xx также могут включать в себя более сложные логические функции. Некоторые из них представлены на

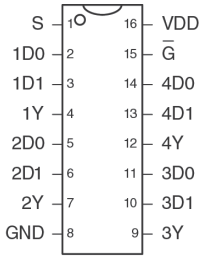
рис. А.2 и **А.3**. Такие микросхемы называются *средними интегральными схемами* (СИС). Большинство из них использует более крупные корпуса для размещения большего количества входов и выходов. Выводы питания и земли по-прежнему находятся в правом верхнем и левом нижнем углах каждой микросхемы соответственно. Общее функциональное описание прилагается к каждой микросхеме. Более полная информация о каждой микросхеме приведена в технической документации производителя.



74153 4 :1 Mux

Two 4:1 Multiplexers
$D_{3:0}$: data
$S_{1:0}$: select
Y: output
Gb: enable

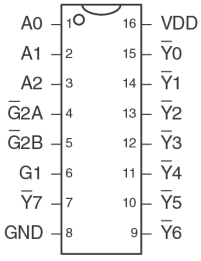
```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```



74157 2 :1 Mux

Four 2:1 Multiplexers
$D_{1:0}$: data
S: select
Y: output
Gb: enable

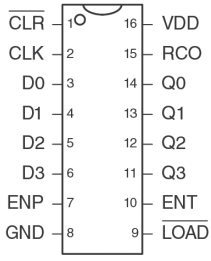
```
always_comb
  if  (Gb) 1Y = 0;
  else     1Y = S ? 1D[1] : 1D[0];
  if  (Gb) 2Y = 0;
  else     2Y = S ? 2D[1] : 2D[0];
  if  (Gb) 3Y = 0;
  else     3Y = S ? 3D[1] : 3D[0];
  if  (Gb) 4Y = 0;
  else     4Y = S ? 4D[1] : 4D[0];
```



74138 3 :8 Decoder

3:8 Decoder
$A_{2:0}$: address
$Yb_{7:0}$: output
G1: active high enable
G2: active low enables

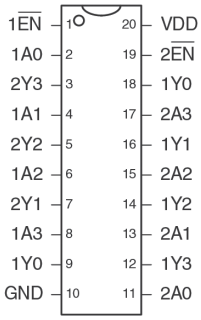
G1	$\overline{G2A}$	$\overline{G2B}$	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



74161/163 Counter

4-bit Counter
CLK: clock
$Q_{3:0}$: counter output
$D_{3:0}$: parallel input
CLRb: async reset (161)
sync reset (163)
LOADb: load Q from D
ENP, ENT: enables
RCO: ripple carry out

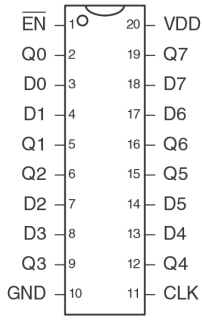
```
always @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```



74244 Tristate Buffer

8-bit Tristate Buffer
$A_{3:0}$: input
$Y_{3:0}$: output
ENb: enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```



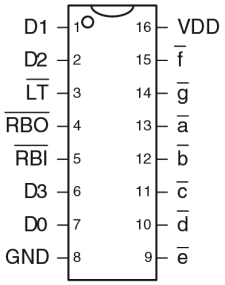
74377 Register

8-bit Enableable Register
CLK: clock
$D_{7:0}$: data
$Q_{7:0}$: output
ENb: enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Замечание: имена переменных в SystemVerilog не могут начинаться с цифр, но имена в примерах кода на рис. А.2 выбраны так, чтобы соответствовать спецификации производителя.

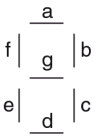
Рис. А.2 **Средние интегральные схемы**



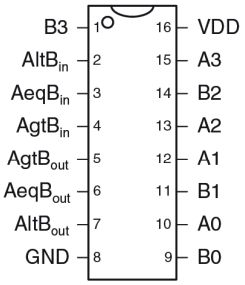
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

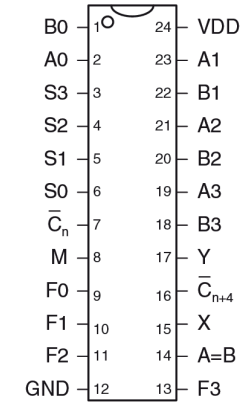


7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always_comb
   if (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```



74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A       :  A                  + ~Cbn;
    0001: Y = M ? ~(A | B) :  A       +  B       + ~Cbn;
    0010: Y = M ? (~A) & B :  A       + ~B       + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111            + ~Cbn;
    0100: Y = M ? ~(A & B) :  A       + (A & ~B) + ~Cbn;
    0101: Y = M ? ~B       : (A | B)  + (A & ~B) + ~Cbn;
    0110: Y = M ? A ^ B    :  A       - B        -  Cbn;
    0111: Y = M ? A & ~B   : (A & ~B)            -  Cbn;
    1000: Y = M ? ~A + B   :  A       + (A & B)  + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A       +  B       + ~Cbn;
    1010: Y = M ? B        : (A | ~B) + (A & B)  + ~Cbn;
    1011: Y = M ? A & B    : (A & B)             + ~Cbn;
    1100: Y = M ? 1        :  A       +  A       + ~Cbn;
    1101: Y = M ? A | ~B   : (A | B)  +  A       + ~Cbn;
    1110: Y = M ? A | B    : (A | ~B) +  A       + ~Cbn;
    1111: Y = M ? A        :  A                  -  Cbn;
  endcase
```

Рис. А.3 **Другие средние интегральные схемы (СИС)**

A.3. Программируемая логика

Программируемая логика состоит из матриц элементов, которые можно сконфигурировать для выполнения определенных логических функций. Мы уже описали три вида программируемой логики: программируемое

постоянное запоминающее устройство (PROM), программируемые логические матрицы (PLA) и программируемая пользователем вентильная матрица (FPGA). Данный раздел описывает реализацию микросхем каждого из этих трех видов.

Конфигурирование в таких микросхемах может быть осуществлено пережиганием находящихся в микросхемах перемычек для соединения/разъединения элементов схемы. Это так называемая *однократно программируемая логика* (OTP), поскольку после того, как перемычка разрушена, ее невозможно восстановить. Альтернативным решением является сохранение конфигурации в памяти микросхемы, которую можно перепрограммировать много раз. Перепрограммируемая логика удобна в лаборатории, поскольку одну и ту же схему можно повторно использовать в процессе разработки разных устройств.

A.3.1. PROM

Как обсуждалось в **разделе 5.5.7**, блоки PROM могут быть использованы в качестве таблицы преобразования. PROM емкостью 2^N слов $\times M$ бит можно запрограммировать для выполнения любой комбинационной функции с N входами и M выходами. Изменения во время разработки представляют собой простую замену содержимого блока PROM, а не переконфигурирование соединений между микросхемами. Таблицы преобразования полезны для реализации простых функций, но с ростом количества входов становятся недопустимо дорогими.

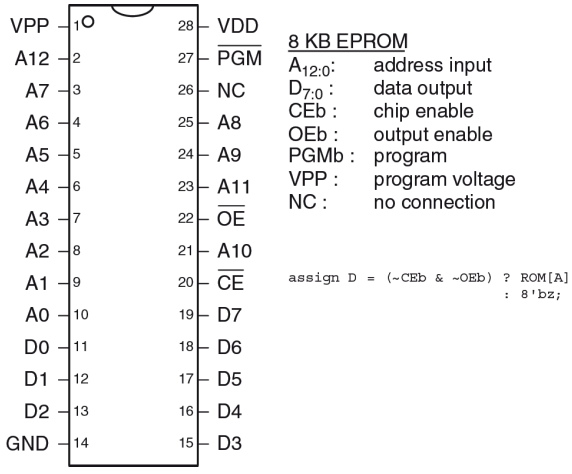


Рис. А.4 ЭСППЗУ 2764 на 8 Кбайт

Например, классическое стираемое ППЗУ (EPROM) 2764 на 8 Кбайт (64 кбит) показано на **рис. А.4**. EPROM имеет 13 адресных входов для

выбора одного из 8K слов и 8 выводов данных для считывания байта информации из выбранного слова. Выводы output enable и chip enable должны активироваться совместно для чтения данных.

Максимальная задержка распространения сигнала в такой микросхеме составляет 200 пикосекунд. При нормальном функционировании выводы $\overline{PGM} = 1$ и вывод VPP не используется. EPROM обычно программируется через специальный программатор, который устанавливает $\overline{PGM} = 0$, подает напряжение 13 В на вывод VPP и использует специальную последовательность входных сигналов для конфигурирования содержимого памяти.

Современные PROM основаны на тех же принципах, но имеют значительно большую емкость и больше выводов. Флеш-память – самый дешевый вариант PROM, продававшийся по цене примерно $1 за гигабайт в 2012 году. Цены традиционно снижаются на 30–40 % в год.

A.3.2. Блоки PLA

Как уже обсуждалось в **разделе 5.6.1**, PLA (программируемые логические матрицы) содержат массивы элементов И и ИЛИ для вычисления любых комбинационных функций, предоставленных в совершенной дизъюнктивной нормальной форме. Массивы элементов И и ИЛИ могут быть запрограммированы по той же технологии, что и PROM. Матрицы PLA содержат два столбца для каждого входа и один столбец для каждого выхода, а также одну строку для каждого минтерма. Такая структура более эффективна, чем PROM для многих функций, но матрицы все равно становятся слишком большими для реализации сложных функций с большим количеством входов/выходов и минтермов.

Многие производители расширили базовую концепцию PLA до уровня программируемых логических устройств (PLD), которые включают в себя регистры. Микросхема 22V10 – одна из наиболее распространенных PLD. Она содержит 12 выделенных входов и 10 выходов. Выходы могут подключаться напрямую к PLA или к тактируемым регистрам микросхемы. Также выходы можно подключать обратно ко входам PLA. Таким образом, с помощью микросхемы 22V10 можно непосредственно создавать конечный автомат с 12 входами, 10 выходами и 10 битами состояния. Микросхема 22V10 стоит примерно $2 в партиях от 100 штук. PLD стали устаревшей технологией из-за быстрого увеличения емкости и снижения цены на FPGA.

A.3.3. FPGA

Как уже говорилось в **разделе 5.6.2**, FPGA состоят из массивов конфигурируемых логических элементов, также называемых *конфигурируемыми логическими блоками* (КЛБ), соединенных друг с другом программируемыми межсоединениями. Логические элементы состоят

из небольших таблиц преобразования и триггеров. FPGA изящно масштабируются до больших размерностей с тысячами таблиц преобразования. Xilinx и Altera – два ведущих производителя FPGA.

Таблицы преобразования и программируемые межсоединения позволяют создать практически любую логическую функцию. Но они на порядок менее эффективны по показателю скорости и себестоимости (на единицу площади микросхемы), чем аппаратно реализованные версии тех же функций. Поэтому FPGA часто включают в себя специальные блоки – память, умножители и даже целые микропроцессоры.

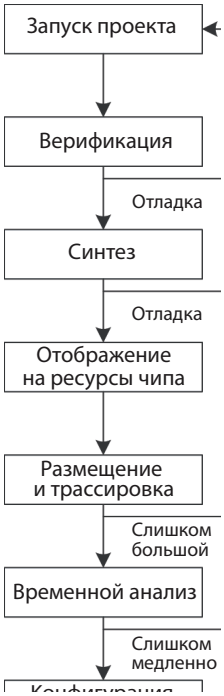


Рис. А.5 Процесс разработки цифровой системы на FPGA

На **рис. А.5** показан процесс разработки цифровой системы на FPGA. Разработка обычно ведется на языке описания аппаратуры (HDL), хотя некоторые среды разработки FPGA поддерживают графический ввод схем. После разработки проводится моделирование. На входы схемы подаются тестовые воздействия, и выходные сигналы сравниваются с ожидаемыми, чтобы *проверить* работу устройства. Обычно после моделирования требуется отладка схемы и повторное моделирование. Далее в процессе логического *синтеза* HDL-описание преобразуется в логические функции. Современные средства синтеза строят принципиальные схемы реализаций этих функций, и разработчик анализирует эти схемы, а также все предупреждения, появившиеся в процессе синтеза, чтобы убедиться, что реализована именно желаемая логика.

Иногда небрежное описание приводит к генерации схем, бóльших по размерам, чем требуется, и схем с асинхронной логикой работы. Когда результаты синтеза удовлетворительны, среда разработки *отображает* функции на логические элементы конкретной микросхемы. Инструмент *размещения и трассировки* определяет, к какой таблице преобразования относится каждая функция и как эти таблицы соединены между собой. Задержка распространения сигнала возрастает с увеличением длины пути прохождения сигнала, поэтому наиболее ответственные цепи следует размещать как можно ближе друг к другу. Если проект слишком велик, чтобы уместиться на одной микросхеме FPGA, его придется перерабатывать заново. *Временной анализ* сравнивает ограничения во временной области (например, может быть задано, что схема должна работать на тактовой частоте 100 МГц) с реальными задержками, получаемыми в схеме, и выдает отчет об ошибках. Если схема работает слишком медленно, ее, возможно, следует модернизировать или применить метод ковейеризации. После того как проект исправлен, генерируется файл, определяющий содержание каждого логического элемента и конфигурацию всех трассировочных ресурсов или межсоединений в FPGA. Многие FPGA сохраняют *конфигурационную* информацию в статическом ОЗУ, которое должно

перезагружаться каждый раз при включении FPGA. FPGA может загружать эту информацию с лабораторного компьютера или считывать ее из энергонезависимой памяти при подаче напряжения питания.

Пример A1 АНАЛИЗ ВРЕМЕННЫХ ДИАГРАММ FPGA

Алиса П. Хакер использует FPGA для реализации сортировщика драже M&M с датчиком цвета и моторами, чтобы складывать красные конфеты в одну банку, а зеленые – в другую. Ее разработка представляет собой конечный автомат, и она использует FPGA Cyclone IV GX. Согласно технической документации, данная FPGA имеет временные характеристики, представленные в **табл. A.1**.

Таблица A.1 Временные характеристики Cyclone IV GX

Величина	Значение (пс)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (на один логический элемент)	381
t_{wire} (между логическими элементами)	246
t_{skew}	0

Алиса хотела бы, чтобы ее конечный автомат работал на скорости 100 МГц. Каково максимальное количество логических элементов модет быть в критическом пути такого автомата? Какова максимальная скорость, на которой сможет работать конечный автомат?

Решение На частоте 100 МГц время цикла T_c составляет 10 нс. Алиса использует **уравнение (3.13)** для получения минимальной задержки распространения t_{pd} на комбинационных элементах за время цикла:

$$t_{pd} \leq 10 \text{ нс} - (0{,}199 \text{ нс} + 0{,}076 \text{ нс}) = 9{,}725 \text{ нс}. \quad \text{(A.1)}$$

С учетом задержек в логических элементах и линиях передач для автомата Алиса может использовать самое большее 15 последовательно включенных логических элементов (9,725 / 0,627) для реализации логики переходов.

Самая высокая скорость работы автомата на FPGA Cyclone IV GX достигается при использовании одного логического элемента для перехода к следующему логическому состоянию. Минимальное время цикла

$$T_c \geq 381 \text{ пс} + 19 \text{ пс} + 76 \text{ пс} = 656 \text{ пс}. \quad \text{(A.2)}$$

Таким образом, максимальная частота составляет 1.5 ГГц.

Фирма Altera выпускала Cyclone IV FPGA, содержавшую 14 400 логических элементов по цене $25 в 2012 году. Большинство FPGA среднего размера обычно стоят несколько долларов. Самые большие FPGA стоят сотни и даже тысячи долларов. Цены снижаются примерно на 30 % в год, и FPGA становятся очень популярными.

А.4. Заказные специализированные интегральные схемы

Заказные интегральные схемы (application specific integrated circuit, ASIC) – это микросхемы, разработанные для специальных применений. Графические ускорители, микросхемы сетевых интерфейсов и микросхемы мобильных телефонов – примеры заказных интегральных микросхем (ИМС). Разработчик подобных схем располагает транзисторы таким образом, чтобы сформировать логические элементы и соединить их друг с другом. Поскольку архитектура заказных ИМС аппаратно фиксирована, они обычно в несколько раз быстрее FPGA и занимают на порядок меньше места на чипе микросхемы и, соответственно, стоят меньше, чем FPGA с такими же функциями. При этом стоимость производства *фотошаблонов* для литографии, определяющих, где должны располагаться транзисторы и проводники на микросхеме, составляет сотни тысяч долларов. Технологический процесс на полупроводниковом производстве обычно занимает от 6 до 12 недель и включает в себя производство, корпусирование и испытания ИМС. Если ошибки обнаружены после производства заказной ИМС, разработчик должен ее исправить, изготовить новые фотошаблоны и дождаться выпуска новой партии микросхем. Таким образом, заказные ИМС применимы только для изделий, производимых массово и функции которых строго определены заранее.

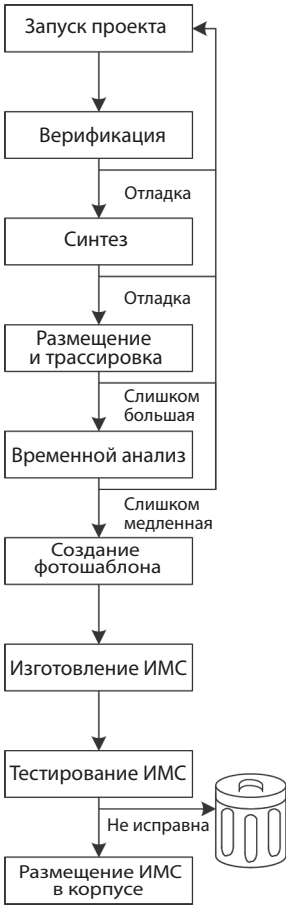


Рис. А.6 Процесс разработки заказной ИМС

На **рис. А.6** показан процесс разработки заказной ИМС, который схож с процессом разработки FPGA, показанным на **рис. А.5**. Логическая верификация проекта особенно важна, поскольку исправление ошибок после производства фотошаблонов – очень дорогостоящее мероприятие. В результате выполнения синтеза получается *список соединений*, или *нетлист*, в котором перечислены логические элементы и соединения между ними. Логические элементы, указанные в этом списке, размещаются на кристалле, между ними прокладываются соединяющие их проводники. После того как проект признается удовлетворительным, создаются фотошаблоны, используемые далее для производства ИМС. Одна-единственная пылинка может испортить всю схему, поэтому микросхемы после производства тестируют.

Доля рабочих микросхем называется *выходом годных изделий* и обычно составляет от 50 до 90 %, в зависимости от размеров микросхем и степени отработанности

производственного процесса. Наконец, рабочие кристаллы микросхемы корпусируются, этот вопрос рассматривается в **разделе A.7**.

A.5. Работа с документацией

Производители микросхем публикуют *техническую документацию* (ТД), в которой описываются назначение и технические характеристики микросхем. Необходимо уметь читать и понимать техническую документацию. Одной из основных причин ошибок при разработке цифровых систем является непонимание того, как работает микросхема.

ТД обычно можно найти на сайте производителя. Если вы не можете найти техническую документацию на компонент или у вас нет полной документации из других источников, не используйте этот компонент. Некоторые разделы в технической документации могут быть непонятны. Часто производители издают справочники, включающие в себя ТД сразу на многие однотипные компоненты. В начале справочника приводится дополнительная пояснительная информация. Эту информацию обычно можно найти в сети Интернет.

В данном разделе рассматривается ТД компании Texas Instruments (TI) на микросхему инвертора 74HC04. Эта ТД довольно проста, но описывает многие основные элементы. Компания TI производит широкий ассортимент микросхем серии 74xx. В прошлом микросхемы этой серии производились многими компаниями, но из-за снижения продаж рынок консолидируется.

На **рис. A.7** показана первая страница ТД. Некоторые основные разделы документа выделены синим. Название ТД – SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS означает, что микросхема содержит шесть инверторов. SN означает компанию производителя (TI). Обозначения других производителей: MC для Motorola и DM для National Semiconductor. В целом можно не обращать внимания на эти обозначения, поскольку все производители выпускают совместимые микросхемы серии 74xx. HC обозначает семейство логических схем (высокоскоростные КМОП). Семейство логических схем определяет быстродействие и энергопотребление микросхемы, но не назначение. Например, микросхемы 7404, 74HC04 и 74LS04 состоят из шести инверторов, но у них разные производительность и стоимость. Остальные семейства логических схем рассмотрены в **разделе A.6**. Микросхемы серии 74xx работают в коммерческом или промышленном температурных диапазонах (от 0 до 70 °C или от −40 до 85 °C соответственно), тогда как микросхемы серии 54xx работают в диапазоне, подходящем для военных применений (от −55 до 125 °C), и стоят дороже. По другим же параметрам эти микросхемы взаимозаменяемые.

Микросхемы 7404 доступны в различных корпусах, при покупке необходимо заказывать ту, которая вам нужна. Тип корпуса определяет-

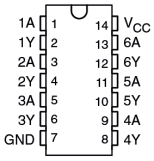
ся суффиксом в обозначении компонента. *N* обозначает *пластмассовый корпус с двухрядным расположением выводов* (PDIP), который устанавливается на макетную плату или может быть впаян в сквозные отверстия на печатной плате. Другие типы корпусов рассматриваются в **разделе А.7**.

SN54HC04, SN74HC04
HEX INVERTERS

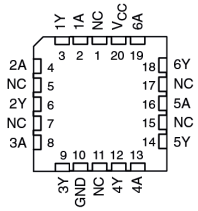
SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- **Wide Operating Voltage Range of 2 V to 6 V**
- **Outputs Can Drive Up To 10 LSTTL Loads**
- **Low Power Consumption, 20-μA Max I_{CC}**
- **Typical tpd = 8 ns**
- **±4-mA Output Drive at 5 V**
- **Low Input Current of 1 μA Max**

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)



SN54HC04 . . . FK PACKAGE
(TOPVIEW)



NC – No internal connection

description/ordering information

The 'HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this data sheet.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

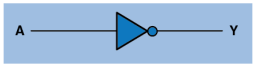
Рис. А.7 Техническая документация на микросхему серии 7404, страница 1

Таблица истинности показывает, что каждый вентиль инвертирует свое входное значение. Если на выводе *A* высокий уровень сигнала, то на выводе *Y* – низкий, и наоборот. В данном случае таблица простая, но для сложных микросхем она будет не такой простой.

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)



absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC} ..... –0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1) ..... ±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1) ..... ±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC}) ..... ±25 mA
Continuous current through V_{CC} or GND ..... ±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package ..... 86° C/W
N package ..... 80° C/W
NS package ..... 76° C/W
PW package ..... 131° C/W
Storage temperature range, T_{stg} ..... –65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Рис. А.8 Техническая документация на микросхему серии 7404, страница 2

На **рис. А.8** показана вторая страница ТД. Условное обозначение показывает, что микросхема состоит из инверторов. Раздел *absolute maximum* (абсолютные максимумы) определяет условия, при нарушении которых микросхема может быть разрушена. В частности, напряжение питания V_{CC} (в этой книге обозначается также как V_{DD}) не должно превышать 7 В. Постоянный выходной ток не должен превышать 25 мА. *Тепловое сопротивление*, или импеданс, θ_{JA}, используется при расчете повышения температуры из-за рассеяния мощности микросхемой.

Если *температура окружающей среды* T_A вокруг микросхемы и микросхема рассеивает мощность P_{chip}, то температура кристалла микросхемы в месте *соединения* с корпусом равна

$$T_j = T_A + P_{\text{chip}}\theta. \qquad \text{(A.3)}$$

Например, если микросхема 7404 в пластиковом DIP-корпусе работает в нагретом устройстве при температуре 50 °C и потребляет 20 мВт, температура *соединения* кристалла с корпусом поднимется до 50 °C + 0,02 Вт × 80 °C/Вт = 51,6 °C. Внутреннее рассеяние мощности редко имеет значение для микросхем серии 74xx, но оно становится важным для современных микросхем, которые рассеивают мощность в десятки ватт и больше.

Раздел *recommended operating conditions* (рекомендуемые условия работы) определяет, в каких условиях должна работать микросхема. При работе в таких условиях параметры микросхемы должны соответствовать техническим условиям. Эти условия более строгие, чем предельные значения. Например, напряжение питания должно быть между 2 и 6 В. Уровень входного сигнала логической единицы семейства микросхем HC зависит от напряжения V_{DD}. При V_{DD} = 5 В для логической 1 следует подавать на вход напряжение 4,5 В, чтобы предусмотреть 10%-ное падение напряжения питания, вызванное шумами в системе.

На **рис. А.9** показана третья страница ТД. Раздел *electrical characteristics* (электрические характеристики) описывает поведение микросхемы при работе в рекомендуемых режимах работы при постоянном напряжении на входах.

Например, если V_{CC} = 5 В (и может снизиться до 4,5 В) и выходной ток I_{OH}/I_{OL} не превышает 20 мкА, то V_{OH} = 4,4 В и V_{OL} = 0,1 В в наихудшем случае. Если выходной ток возрастет, то разница между выходными напряжениями V_{OH} и V_{OL} уменьшается, поскольку транзисторы в микросхеме с трудом могут обеспечивать такой ток. Семейство HC использует КМОП-транзисторы, потребляющие малые токи. Ток на каждом входе гарантированно меньше 1000 нА и обычно порядка 0,1 нА при комнатной температуре. *Ток покоя* (I_{DD}), потребляемый, когда микросхема находится в статическом режиме, составляет менее 20 мкА. Каждый вход имеет емкость менее 10 пФ.

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	I_{OH} = –20 µA	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		I_{OH} = –4 mA	4.5 V	3.98	4.3		3.7		3.84		
		I_{OH} = –5.2 mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	I_{OL} = 20 µA	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		I_{OL} = 4 mA	4.5 V		0.17	0.26		0.4		0.33	
		I_{OL} = 5.2 mA	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, I_O = 0		6 V			2		40		20	µA
C_I			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, T_A = 25 °C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Рис. A.9 Техническая документация на микросхему серии 7404, страница 3

Раздел *switching characteristics* (характеристики переключения) определяет, как микросхема ведет себя при рекомендованных режимах работы при изменении состояний входов. *Задержка распространения*

сигнала, t_{pd}, измеряется от момента перехода входного сигнала через 0,5 V_{CC} до момента перехода выходного сигнала через 0,5 V_{CC}. Если V_{CC} обычно 5 В и емкость нагрузки микросхемы составляет менее 50 пФ, задержка распространения не будет превышать 24 нс (обычно она даже меньше). Следует помнить, что каждый вход может иметь емкость 10 пФ, поэтому микросхема не может управлять более чем пятью одинаковыми микросхемами на максимальной скорости. Разумеется, паразитная емкость проводников, соединяющих микросхемы, уменьшает возможную полезную нагрузку. *Время переключения* (время нарастания/спада сигнала) измеряется как время между значениями 0,1 V_{CC} и 0,9 V_{CC}.

В **разделе 1.8** говорилось, что микросхемы потребляют *статическую* и *динамическую* мощность. Статическая мощность у микросхем семейства HC невысока. При температуре 85 °C максимальный ток покоя составляет 20 мкА. При напряжении 5 В статическая мощность составляет 0,1 мВт. Динамическая мощность зависит от емкости нагрузки и частоты переключения. Микросхемы серии 7404 имеют внутреннюю емкость (учитываемую при расчете рассеиваемой мощности) 20 пФ на инвертор. Если все шесть инверторов микросхемы серии 7404 переключаются с частотой 10 МГц и управляют внешними нагрузками по 25 пФ, то динамическая мощность согласно **уравнению (1.4)** равна ½(6)(20 пФ + 25 пФ)(5^2)(10 МГц) = 33,75 мВт и максимальная полная мощность составляет 33,85 мВт.

A.6. Семейства логических микросхем

Микросхемы серии 74xx производятся при помощи различных технологий, определяемых *семействами логических микросхем*, которые имеют различные быстродействие, мощность и логические уровни. Другие микросхемы обычно совместимы с некоторыми семействами этих микросхем. Оригинальные микросхемы, например 7404, производились на основе биполярных транзисторов при помощи технологии *транзисторно-транзисторной логики* (ТТЛ). Современные технологии добавляют одну или несколько букв в маркировку после цифр 74 для отображения семейства, например 74LS04, 74HC04 или 74AHCT04. В **табл. А.2** приведены наиболее распространенные семейства микросхем с напряжением питания 5 В.

Достижения в биполярных схемах и развитие технологий привели к созданию семейств на основе транзисторно-транзисторной логики с диодами Шоттки (ТТЛШ) (*S*) и маломощной транзисторно-транзисторной логики с диодами Шоттки (*LS*). Оба семейства работают быстрее ТТЛ. ТТЛШ потребляют большую мощность, а маломощные ТТЛШ – меньшую. Усовершенствованные ТТЛШ (AS) и усовершенствованные маломощные ТТЛШ (ALS) семейства микросхем имеют улучшенные ха-

рактеристики быстродействия и мощности по сравнению с семействами S и LS. Микросхемы семейства быстрой логики (F) работают быстрее и потребляют меньше мощности, чем AS.

Табл. A.2. Характеристики семейств логики с напряжением 5 В

Характеристики	Биполярные/ТТЛ						КМОП		КМОП/ТТЛ-совместимые	
	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (нс)	22	9	12	7,5	10	6	21	7,5	30	7,7
V_{IH} (В)	2	2	2	2	2	2	3,15	3,15	2	2
V_{IL} (В)	0,8	0,8	0,8	0,8	0,8	0,8	1,35	1,35	0,8	0,8
V_{OH} (В)	2,4	2,7	2,7	2,5	2,5	2,5	3,84	3,8	3,84	3,8
V_{OL} (В)	0,4	0,5	0,5	0,5	0,5	0,5	0,33	0,44	0,33	0,44
I_{OH} (мА)	0,4	1	0,4	2	0,4	1	4	8	4	8
I_{OL} (мА)	16	20	8	20	8	20	4	8	4	8
I_{IL} (мА)	1,6	2	0,4	0,5	0,1	0,6	0,001	0,001	0,001	0,001
I_{IH} (мА)	0,04	0,05	0,02	0,02	0,02	0,02	0,001	0,001	0,001	0,001
I_{DD} (мА)	33	54	6,6	26	4,2	15	0,02	0,02	0,02	0,02
C_{Pd} (пФ)	Неизвестно						20	12	20	14
Стоимость* ($ США)	Устаревшие	0,63	0,25	0,53	0,32	0,22	0,12	0,12	0,12	0,12

* Стоимость за микросхемы серии 7408 от Texas Instruments в 2012 году (за 1000 шт.).

Все микросхемы этих семейств обеспечивают больший ток для выходов с низким уровнем сигнала, чем с высоким, и поэтому имеют несимметричные логические уровни. Они согласовываются с логическими уровнями ТТЛ: $V_{IH} = 2$ В, $V_{IL} = 0{,}8$ В, $V_{OH} > 2{,}4$ В и $V_{OL} < 0{,}5$ В.

После того как в 1980-х и 1990-х годах были разработаны КМОП-микросхемы, они стали широко распространены благодаря малому энергопотреблению и малому входному току. Семейства микросхем на базе высокоскоростной КМОП (HC) и усовершенствованной высокоскоростной КМОП (AHC) технологии практически не потребляют статической мощности. Также они обеспечивают одинаковые токи на выходах с высоким и низким уровнями сигнала. Они согласовываются с логическими уровнями КМОП: $V_{IH} = 3{,}15$ В, $V_{IL} = 1{,}35$ В, $V_{OH} > 3{,}8$ В и $V_{OL} < 0{,}44$ В. К сожалению, эти логические уровни не совместимы с уровнями ТТЛ-схем, поскольку высокий уровень выходного сигнала ТТЛ 2,4 В может быть не распознан как высокий уровень входного сигнала КМОП. Это служит причиной использования высокоскоростной ТТЛ-совместимой КМОП (HCT) технологии и усовершенствованной высокоскоростной ТТЛ-совместимой КМОП (AHCT) технологии. Микросхемы, выполненные по этим технологиям, принимают входные логические уровни ТТЛ и формируют выходные логические уровни КМОП. Микросхемы этих семейств работают немного медленнее, чем их КМОП-аналоги. Все КМОП-микросхемы чувствительны к электростатическим разрядам (ESD), вызывае-

мым статическим электричеством. Перед работой с КМОП-микросхемами обязательно следует заземляться[1], иначе можно их испортить.

Стоимость микросхем серии 74xx невысока. Микросхемы новых серий часто бывают дешевле, чем устаревшие. Микросхемы семейства LS широко распространены и имеют высокую надежность, благодаря чему подходят для лабораторных и домашних проектов, где не предъявляются специальные требования по производительности.

Микросхемы со стандартом напряжения питания 5 В стали исчезать в середине 1990-х, когда транзисторы стали настолько маленькими по размерам, что не могли выдерживать такое напряжение. Более того, низкое напряжение влечет за собой более низкое энергопотребление. Сегодня используются напряжения 3,3; 2,5; 1,8; 1,2 В и даже ниже. Разнообразие питающих напряжение микросхем вызывает трудности согласования микросхем с различными источниками питания. В **табл. А.3** перечислены некоторые семейства низковольтных микросхем. Не все компоненты серии 74xx доступны в каждом семействе.

Таблица А.3 Характеристики семейств низковольтной логики

	LVC			ALVC			AUC		
V_{dd} (В)	3,3	2,5	1,8	3,3	2,5	1,8	2,5	1,8	1,2
t_{pd} (нс)	4,1	6,9	9,8	2,8	3	?*	1,8	2,3	3,4
V_{IH} (В)	2	1,7	1,17	2	1,7	1,17	1,7	1,17	0,78
V_{IL} (В)	0,8	0,7	0,63	0,8	0,7	0,63	0,7	0,63	0,42
V_{OH} (В)	2,2	1,7	1,2	2	1,7	1,2	1,8	1,2	0,8
V_{OL} (В)	0,55	0,7	0,45	0,55	0,7	0,45	0,6	0,45	0,3
I_O (мА)	24	8	4	24	12	12	9	8	3
I_I (мА)	0,02			0,005			0,005		
I_{DD} (мА)	0,01			0,01			0,01		
C_{pd} (пФ)	10	9,8	7	27,5	23	?*	17	14	14
Стоимость ($ США)	0,17			0,20			не доступно		

* Задержка распространения и емкость не доступны на момент создания книги.

Все низковольтные семейства микросхем построены на КМОП-транзисторах, основе современных интегральных схем. Они работают в широком диапазоне напряжений V_{DD}, но их скорость работы падает при низком напряжении. Микросхемы семейств низковольтных КМОП (*LVC*) и усовершенствованных низковольтных КМОП (*ALVC*) обычно используются при напряжениях 3,3; 2,5 или 1,8 В. *LVC*-микросхемы выдерживают входное напряжение до 5,5 В, поэтому могут использоваться совместно с микросхемами КМОП и ТТЛ стандарта 5 В. Микросхемы семейства усовершенствованных ультранизковольтных КМОП (*AUC*) обычно используют напряжения 2,5; 1,8 или 1,2 В и работают очень быстро. Микросхемы ALVC и AUC выдерживают входное напряжение до 3,6 В, поэтому их можно использовать с микросхемами стандарта 3,3 В.

[1] Используя, например, специальный браслет. – *Прим. перев.*

В микросхемах FPGA обычно присутствуют отдельные напряжения питания для внутренней логики (называемой ядром) и для входных/выходных выводов. По мере развития FPGA напряжение питания ядра снизилось с 5 до 3,3; 2,5; 1,8 и 1,2 В для экономии энергии и чтобы избежать выхода из строя очень маленьких по размерам транзисторов. FPGA имеют конфигурируемые входы/выходы, которые могут работать при различных напряжениях для обеспечения совместимости с остальными элементами системы.

A.7. Корпуса и монтаж интегральных схем

Интегральные схемы чаще всего помещаются в пластиковые или керамические корпуса. Корпуса выполняют несколько функций: соединение крошечных металлических выводов кристалла с большими выводами корпуса для простоты подключения, защита микросхемы от физических повреждений и рассеяние выделяемого микросхемой тепла на большей поверхности для охлаждения. Корпуса размещаются на макетной плате или печатной плате и соединяются проводниками в систему.

Корпуса

На **рис. A.10** показаны разнообразные корпуса интегральных схем. Корпуса могут быть разделены на две большие группы: *корпуса для монтажа в отверстия* и *корпуса поверхностного монтажа* (SMT). Корпуса для монтажа в отверстия, как следует из их названия, имеют выводы, которые могут быть вставлены в отверстия в печатной плате или в гнездо. *Корпуса с двухрядным расположением выводов* (DIP) имеют два ряда выводов с расстоянием между выводами 2,54 мм. *Корпус с матрицей стержневых выводов* (PGA) содержит больше выводов в меньшем корпусе благодаря расположению выводов под корпусом. SMT-корпуса припаиваются непосредственно к поверхности печатной платы, а не в отверстия.

Контакты SMT микросхем называются *выводами* (pin). *Тонкий малогабаритный корпус* (TSOP) состоит из двух рядов близко расположенных выводов (обычно расстояние между выводами составляет 0,5 мм). *Пластмассовые кристаллоносители* (PLCC) имеют J-образные выводы со всех четырех сторон, расстояние между выводами 1,27 мм. Их можно припаивать непосредственно на плату или размещать в специальных гнездах. Плоский *корпус с четырехсторонним расположением выводов* (QFP) содержит большее количество контактов благодаря близкому расположению выводов с четырех сторон. *Корпуса с матрицей шариковых контактов* (BGA) полностью исключают использование обыч-

ных выводов. Вместо этого у них сотни маленьких шариков припоя на нижней стороне корпуса. Они тщательно размещаются на контактные площадки на печатной плате, затем нагреваются – припой плавится и соединяет корпус микросхемы с платой.

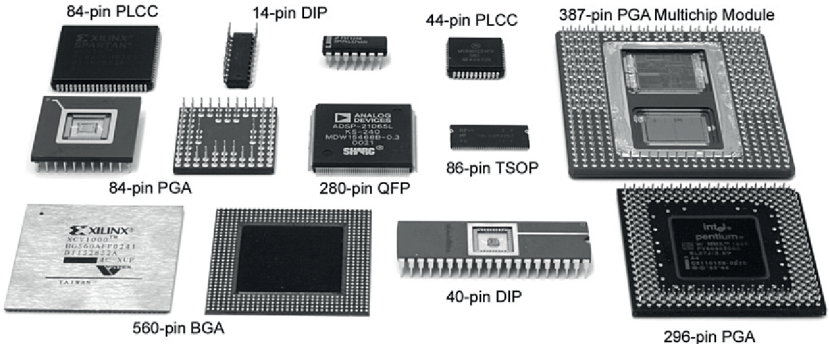


Рис. А.10 Корпуса интегральных схем

Макетные платы

Корпуса DIP удобно использовать для макетирования, поскольку их легко устанавливать на макетные платы. Макетная плата – пластмассовая плата, содержащая ряды гнезд, как показано на **рис. А.11**.

Все пять отверстий в одном ряду соединены друг с другом. Каждый контакт корпуса вставляется в отверстие в отдельном ряду. В соседние отверстия в том же ряду можно подсоединить проводники для соединения контактов. В макетных платах часто есть отдельные столбцы объединенных отверстий вверху платы для распределения питания и земли.

На **рис. А.11** показана макетная плата с мажоритарным вентилем, построенным на базе элементов И микросхемы 74LS08 и элементов ИЛИ микросхемы 74LS32.

Схема представлена на **рис. А.12**. Все элементы на схеме помечены номером микросхемы (08 или 32) и номером контактов входов и выходов (**рис. А.1**). Обратите внимание, что те же самые соединения выполнены на макетной плате.

Входы соединены с контактами 1, 2 и 5 микросхемы 08, выходные сигналы снимаются с контакта 6 микросхемы 32. Питание и земля подаются на выводы 14 и 7 каждой микросхемы соответственно с вертикальных столбцов отверстий с линий питания и общего провода, которые подсоединены к разъемам Va и Vb. Разметка схемы ярлычками и проверка соединений – хороший способ избежать ошибок при макетировании.

К сожалению, в данном случае легко подключить проводник не к тому отверстию, или проводник может выпасть, поэтому макетирование требует концентрированного внимания и обычно отладки в лабора-

тории. Макетные платы пригодны только для прототипирования, а не для производства.

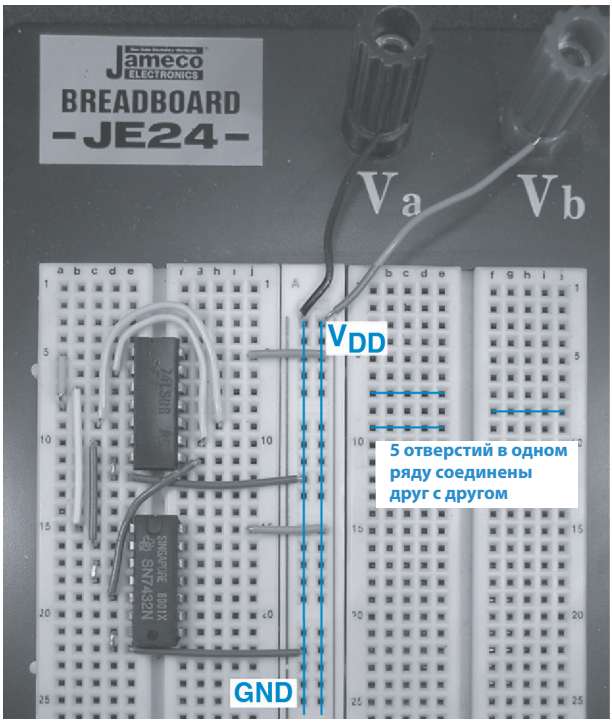


Рис. A.11 Схема мажоритарного вентиля на макетной плате

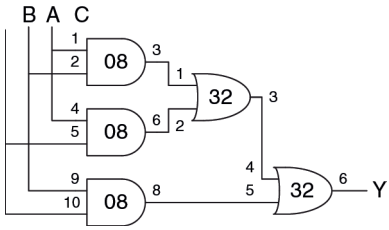


Рис. A.12 Схема мажоритарного вентиля с номерами микросхем и выводов

Печатные платы

Вместо макетных плат микросхемы можно припаивать на *печатные платы*. Печатные платы состоят из чередующихся слоев проводящей меди и изолирующей эпоксидной смолы. Медь вытравливается для формирования проводящих дорожек. Отверстия, называемые переходными, просверливают сквозь плату и металлизируют, чтобы соединить слои между собой. Печатные платы обычно разрабатываются в системах автоматизированного проектирования (САПР/CAD-программы). Можно протравить и просверлить свою плату в лаборатории, или можно отправить проект платы на специализированное предприятие для недорогого массового производства. Предприятия имеют оборотное время длительностью несколько дней (или недель для дешевых партий товаров массового производства) и обычно выставляют счет в несколько сотен долларов на запуск производства и в несколько долларов за каждую плату средней сложности при массовом выпуске.

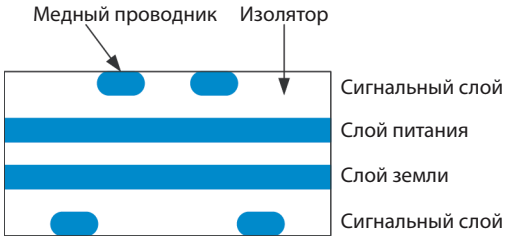


Рис. А.13 Печатная плата в разрезе

Трассировка печатных плат обычно выполняется медью из-за ее низкой стоимости. Проводящие дорожки выполняются на изолирующем материале, обычно зеленом огнеупорном пластике FR4. Также в печатных платах есть медные слои питания и общего провода (земли), расположенные между сигнальными слоями. На **рис. А.13** показана печатная плата в разрезе. Сигнальные слои расположены сверху и снизу, слои питания и земли расположены посередине платы. Слои питания и земли имеют низкое сопротивление и поэтому равномерно распределяют мощность по компонентам платы. Также они делают емкость и индуктивность проводящих дорожек одинаковой и предсказуемой.

На **рис. А.14** показана печатная плата компьютера Apple II+ 1970-х годов. На ней стоит микропроцессор 6502. Ниже расположены 6 микросхем ROM на 16 Кбит, образующих 12 Кбайт памяти ROM, хранящей операционную систему. Три ряда из восьми микросхем DRAM на 16 Кбит обеспечивают 48 Кбайт памяти RAM. Справа расположены несколько рядов микросхем серии 74xx для дешифрации адресов памяти и других

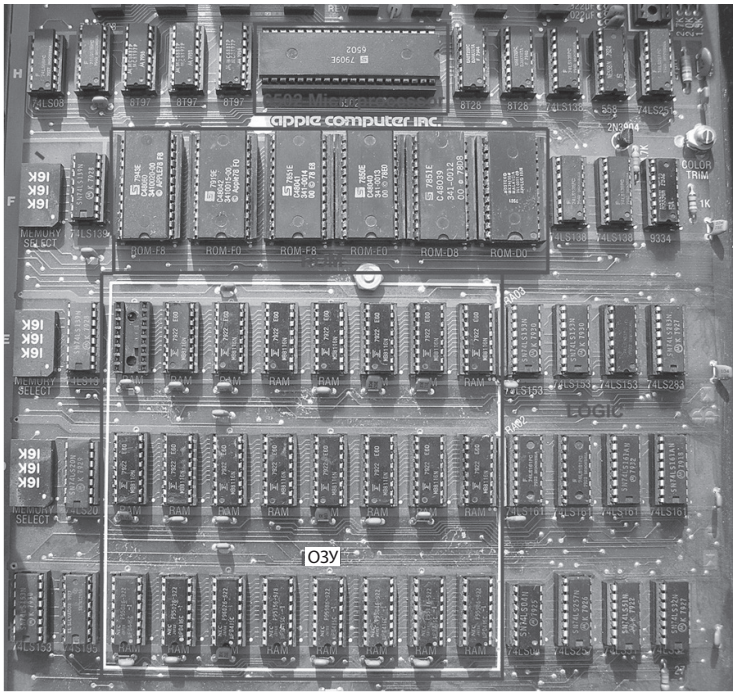


Рис. А.14 Печатная плата компьютера Apple II+

целей. Линии между микросхемами – проводящие дорожки, соединяющие их. Точки на концах некоторых дорожек – сквозные отверстия, заполненные металлом.

Общий обзор

Большинство современных микросхем с большим количеством выводов собираются в SMT-корпуса, в особенности в QFP и BGA. Для этих корпусов необходимо использовать печатную, а не макетную плату. Работа с корпусами BGA особенно сложна, поскольку необходимо специальное оборудование для сборки. Более того, контактные шарики невозможно проверить вольтметром или осциллографом при отладке в лаборатории, поскольку они спрятаны под корпусом.

Таким образом, разработчик должен учитывать проблему корпусов заранее, чтобы определить, будет ли использована макетная плата для отладки и потребуются ли BGA-компоненты. Профессионалы редко пользуются макетными платами, когда они уверены в правильности соединения микросхем без экспериментирования.

А.8. Линии передачи

Ранее мы полагали, что проводники – это эквипотенциальные соединения, которые имеют одинаковое напряжение по всей длине. На самом деле сигналы распространяются по проводникам со скоростью света в виде электромагнитных волн. Если проводники достаточно короткие, или сигналы изменяются медленно, то допущение эквипотенциальности приемлемо. Если проводник очень длинный или сигнал изменяется быстро, время прохождения сигнала по проводнику становится важным для точного определения задержки цепи. Мы должны моделировать такие проводники как линии передачи, в которых волна напряжения и тока распространяется со скоростью света. Когда волна достигает конца линии, она может отразиться в обратную сторону. Отражение может вызвать шумы и сбои в работе системы, если не приняты меры по его ограничению. Поэтому разработчик цифровых систем должен учитывать поведение линии передачи и точно определять задержку и шумовые эффекты в длинных проводниках.

Электромагнитные волны распространяются со скоростью света в конкретном веществе, что довольно быстро, но не мгновенно.

Скорость света ν зависит от диэлектрической проницаемости ε и магнитной проницаемости μ среды:

$$\nu = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}.$$

Скорость света в свободном пространстве $\nu = c = 3\times10^8$ м/с. Сигналы в печатных платах распространяются со скоростью, примерно вдвое меньшей вышеуказанной, поскольку изоляционный материал FR4 имеет диэлектрическую проницаемость в четыре раза больше, чем воздух. Таким образом, сигналы в печатной плате распространяются со скоростью $1{,}5\times10^8$ м/с, или 15 см/нс. Задержка распространения сигнала по линии передачи длиной l описывается следующей формулой:

$$t_d = \frac{l}{\nu}. \tag{A.4}$$

Волновое сопротивление линии передачи Z_0 (произносится «Z нулевое») – это отношение напряжения к току в волне, распространяющейся по линии: $Z_0 = V/I$. Это *не* сопротивление проводника (хорошая линия передачи в цифровой системе обычно имеет пренебрежимо малое сопротивление). Z_0 зависит от индуктивности и емкости линии (выкладки в **разделе А.8.7**) и обычно имеет значение от 50 до 75 Ом:

$$Z_0 = \sqrt{\frac{L}{C}}. \tag{A.5}$$

На **рис. А.15** показано схематическое изображение линии передачи. Изображение напоминает коаксиальный кабель с внутренним проводником сигналов и внешним заземленным проводником, как в телевизионном кабеле.

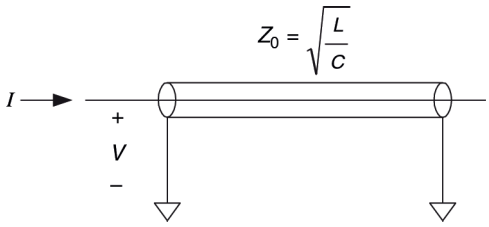


Рис. А.15 Обозначение линии передачи

Ключом к пониманию поведения линии передачи является мысленное представление волны напряжения, распространяющейся вдоль линии со скоростью света. Когда волна достигает конца линии, она может быть поглощена или отражена, в зависимости от оконечного устройства или нагрузки на конце. Отраженные волны распространяются в обратную сторону по линии, накладываясь на уже существующее в линии напряжение. Нагрузка бывает согласованной, холостого хода, короткого замыкания и рассогласованной. В следующих разделах рассматривается распространение волны в линии передачи и что происходит, когда она достигает нагрузки.

А.8.1. Согласованная нагрузка

На рис. А.16 показана линия передачи длиной l с согласованной нагрузкой – это означает, что полное сопротивление нагрузки Z_L равно волновому сопротивлению Z_0.

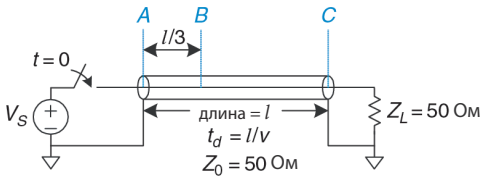


Рис. А.16 Линия передачи с согласованной нагрузкой

Линия передачи имеет волновое сопротивление 50 Ом. Один конец линии подсоединен к источнику напряжения через ключ, который замыкается в момент времени $t = 0$. Другой конец подключен к согласованной нагрузке в 50 Ом. В этом разделе анализируются напряжения и токи в точках A, B и C – в начале линии, на расстоянии в одну треть длины линии и в конце линии соответственно.

На рис. А.17 показаны зависимости напряжения в точках A, B и C от времени. Первоначально в линии нет напряжения и не течет ток, поскольку переключатель разомкнут. В момент времени $t = 0$ ключ замыкается, и источник напряжения генерирует в линии волну с напряжением $V = V_S$. Такая волна называется падающей волной. Поскольку волновое сопротивление составляет Z_0, волна имеет ток $I = V_S / Z_0$. Напряжение достигает начала линии (точка A) сразу, как показано на рис. А.17 (а). Волна распространяется по линии со скоростью света. В момент времени $t_d / 3$ волна достигает точки B. Напряжение в этой точке резко возрастает от 0 до V_S, как показано на рис. А.17 (b). В момент времени t_d падающая волна достигает точки C на конце линии, и там напряжение тоже повышается. Весь ток I течет через сопротивление Z_L, порождая напряжение на сопротивлении $Z_L I = Z_L (V_S / Z_0) = V_S$, поскольку $Z_L = Z_0$.

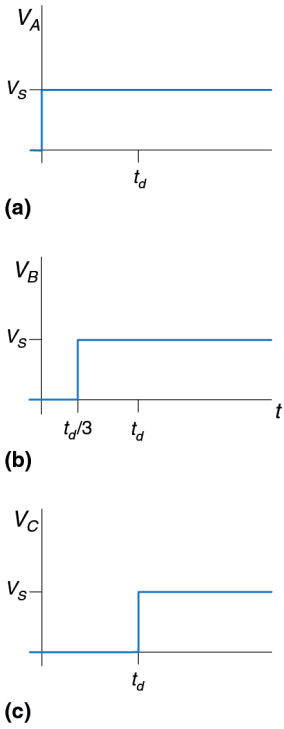


Рис. А.17 Формы напряжения в схеме на рис. А.16 в точках *A*, *B* и *C*

Напряжение соответствует волне, распространяющейся вдоль линии передачи. Таким образом, волна поглощается сопротивлением нагрузки, и линия передачи достигает установившегося режима. В установившемся режиме линия передачи ведет себя как идеальный эквипотенциальный проводник, поскольку по своей сути это только провод. Напряжение во всех точках линии должно быть одинаковым.

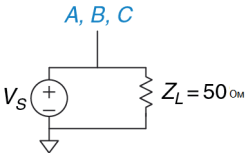


Рис. А.18 Эквивалентная схема линии на рис. А.16 в установившемся режиме

На **рис. А.18** показана эквивалентная схема установившегося режима схемы, представленной на **рис. А.16**. Напряжение равно V_S в каждой точке проводника.

Пример А.2 ЛИНИЯ ПЕРЕДАЧИ С СОГЛАСОВАННЫМ СОПРОТИВЛЕНИЕМ ИСТОЧНИКА И НАГРУЗКИ

На **рис. А.19** показана линия передачи с согласованными сопротивлением источника и нагрузки Z_S и Z_L. Нарисуйте изменения напряжения в точках A, B и C. Когда система достигает установившегося режима, и какова эквивалентная схема установившегося режима?

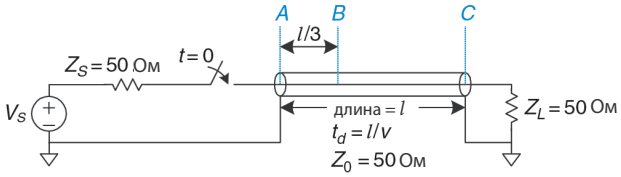


Рис. А.19 Линия передачи с согласованными сопротивлениями источника и нагрузки

Решение Когда источник напряжения имеет сопротивление Z_S, подключенное последовательно с линией передачи, часть напряжения падает на Z_S, и оставшееся напряжение распространяется в линии передачи. Сначала линия передачи ведет себя как сопротивление Z_0, поскольку нагрузка в конце линии не может влиять на поведение линии из-за задержки распространения волны со скоростью света. Таким образом, применяя уравнение делителя напряжения, напряжение падающей волны в линии равно

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2}. \qquad \text{(A.6)}$$

Так, в момент времени $t = 0$ волна напряжения $V = V_S/2$ начинает распространяться в линии из точки A. Как и ранее, сигнал достигает точки B в момент времени $t_d/3$ и точки C в момент времени t_d, как показано на **рис. А.20**. Весь ток поглощается сопротивлением нагрузки Z_L, поэтому схема достигает установившегося режима в момент времени $t = t_d$. В установившемся режиме вся линия находится под напряжением $V_S/2$ – именно так, как можно предсказать по эквивалентной схеме установившегося режима, представленной на **рис. А.21**.

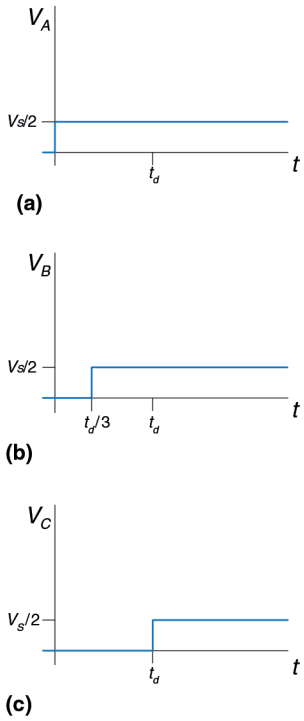


Рис. А.20 Формы напряжений в схеме на рис. А.19 в точках *A*, *B* и *C*

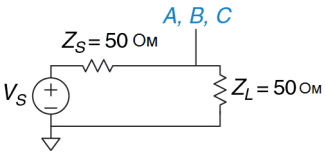


Рис. А.21 Эквивалентная схема линии передачи на рис. А.19 в установившемся режиме

А.8.2. Нагрузка холостого хода

Если сопротивление нагрузки не равно Z_0, нагрузка не может поглощать весь ток, и некоторая часть волны отражается обратно. На **рис. А.22** показана линия передачи с нагрузкой холостого хода. Через такую нагрузку не может течь ток, поэтому ток в точке C всегда должен быть равен 0.

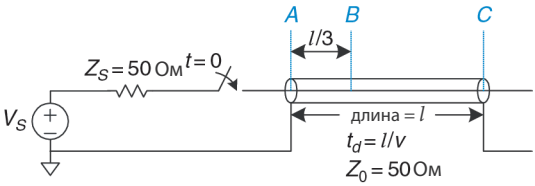


Рис. А.22 Линия передачи с разомкнутым концом

Напряжение в линии изначально равно 0. В момент времени $t = 0$ ключ замыкается и волна напряжения

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2}$$

начинает распространяться в линии.

Следует обратить внимание, что исходная волна такая же, как в **примере А.2**, и она не зависит от оконечного устройства, поскольку нагрузка на конце линии не может влиять на ее поведение в начале линии, по крайней мере пока не пройдет время $2t_d$. Эта волна достигает точки B в момент времени $t_d/3$ и точки C в момент времени t_d, как показано на **рис. А.23**. Когда падающая волна достигает точки C, она не может распространяться вперед, поскольку провод разомкнут. Вместо этого она должна отразиться обратно к источнику. Отраженная волна также имеет напряжение $V_S/2$, поскольку нагрузка холостого хода полностью отражает волну.

Напряжение в любой точке равно сумме падающей и отраженной волн. В момент времени $t = t_d$ напряжение в точке C равно $V = V_S/2 + V_S/2 = V_S$. Отраженная волна достигает точки B в момент времени $5t_d/3$ и точки A в момент времени $2t_d$. Когда волна достигает точки A, она поглощается сопротивлением источника, которое согласовано с волновым сопротивлением линии. Таким образом, система достигает установившегося состояния в момент времени $t = 2t_d$, и линия передачи становится эквивалентной эквипотенциальному проводнику с напряжением V_S и током $I = 0$.

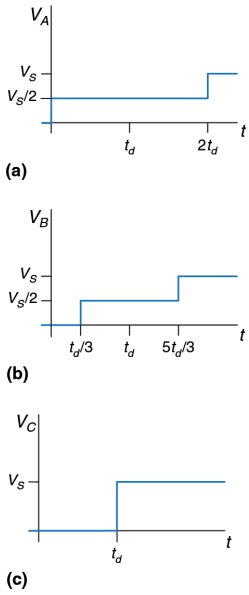


Рис. А.23 Формы напряжений в схеме на рис. А.22 в точках *A*, *B* и *C*

А.8.3. Нагрузка короткого замыкания

На **рис. А.24** показана линия передачи, конец которой короткозамкнут на общий провод (заземлен). Таким образом, напряжение в точке C должно быть всегда равно 0.

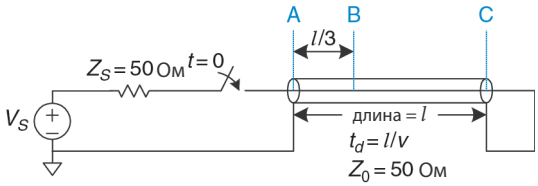


Рис. А.24 Короткозамкнутая линия передачи

Как и в предыдущих примерах, изначально напряжения в линии равны 0. Когда ключ замыкается, волна напряжения $V = V_S/2$ начинает распространяться по линии (**рис. А.25**). Когда волна достигает конца линии, она должна отразиться, поменяв полярность.

Отраженная волна с напряжением $V = -V_S/2$ накладывается на падающую волну, благодаря чему напряжение в точке C остается равным 0. Отраженная волна достигает источника в момент времени $t = 2t_d$ и поглощается сопротивлением источника. В этот момент система достигает установившегося состояния, и линия передачи становится эквивалентна потенциальному проводнику с напряжением $V = 0$.

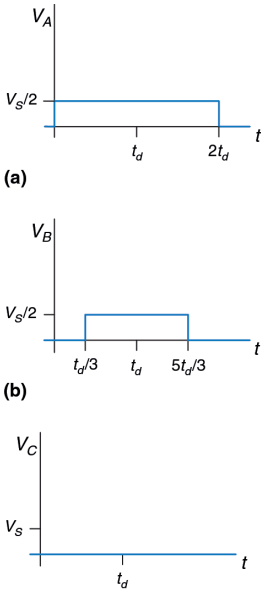


Рис. А.25 Формы напряжений в схеме на рис. А.24 в точках *A*, *B* и *C*

А.8.4. Рассогласованная нагрузка

Сопротивление нагрузки считается рассогласованным, если оно не равно волновому сопротивлению линии. В общем случае, когда падающая волна достигает рассогласованной нагрузки, она частично поглощается и частично отражается. Коэффициент отражения k_r показывает, какая часть падающей волны V_i отразилась: $V_r = k_r V_i$.

В **разделе А.8.8** приводится вывод коэффициента отражения с учетом сохранения величины тока. Также показано, что когда падающая волна, распространяющаяся по линии передачи с волновым сопротивлением Z_0, достигает нагрузки с сопротивлением на конце линии Z_T, коэффициент отражения равен:

$$k = \frac{Z_T - Z_0}{Z_T + Z_0}. \qquad (A.7)$$

Следует обратить внимание на несколько частных случаев. Если цепь на конце разомкнута ($Z_T = \infty$), то $k_r = 1$, потому что падающая волна пол-

ностью отражается (так что ток за пределами линии остается равным нулю). Если на конце цепи короткое замыкание ($Z_T = 0$), то $k_r = -1$, потому что падающая волна отражается с отрицательной полярностью (так что напряжение на конце линии остается равным нулю). Если нагрузка согласована ($Z_T = Z_0$), то $k_r = 0$, потому что падающая волна полностью поглощается.

На **рис. А.26** показаны отражения в линии передачи с рассогласованной нагрузкой с сопротивлением 75 Ом. $Z_T = Z_L = 75$ Ом и $Z_0 = 50$ Ом, поэтому $k_r = 1/5$. Как и в предыдущих примерах, первоначально напряжение в линии равно 0. Когда ключ замыкается, волна напряжения $V = V_S/2$ распространяется по линии, достигая ее конца в момент времени $t = t_d$. Когда падающая волна достигает нагрузки на конце линии, одна пятая волны отражается, а оставшиеся четыре пятых протекают через сопротивление нагрузки. Таким образом, отраженная волна имеет напряжение $V = V_S/2 \times 1/5 = V_S/10$. Суммарное напряжение в точке C состоит из падающего и отраженного напряжений $V_C = V_S/2 + V_S/10 = 3V_S/5$. В момент времени $t = 2t_d$ отраженная волна достигает точки A, где она поглощается согласованной нагрузкой Z_S в 50 Ом. На **рис. А.27** показаны графики токов и напряжений в линии. Следует снова обратить внимание на то, что в установившемся режиме (в данном случае в момент времени $t > 2t_d$) линия передачи эквивалентна эквипотенциальному проводнику, как показано на **рис. А.28**. В установившемся режиме система работает как делитель напряжения, поэтому

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = \left(\frac{75}{75 + 50}\right) = \frac{3V_S}{5}.$$

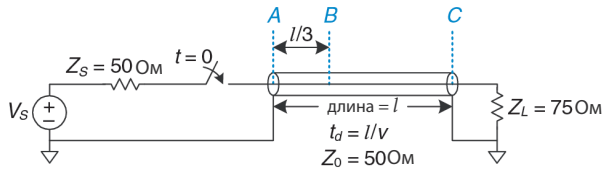


Рис. А.26 Линия передачи с несогласованной нагрузкой

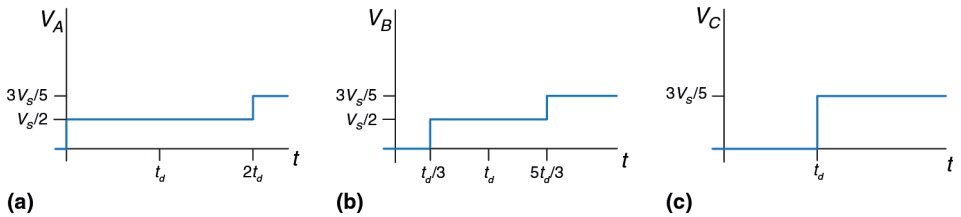


Рис. А.27 Формы напряжений в схеме на рис. А.26 в точках *A*, *B* и *C*

Отражения могут происходить с обеих сторон линии передачи. На **рис. А.29** показана линия передачи с сопротивлением источника Z_S в 450 Ом и разомкнутой цепью на конце. Коэффициенты отражения от нагрузки и источника, k_{rL} и k_{rS}, равны 1 и $4/5$ соответственно. В данном случае волны отражаются от обоих концов линии передачи до достижения установившегося режима.

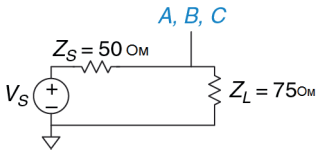


Рис. А.28 Эквивалентная схема линии передачи на рис. А.26 в установившемся режиме

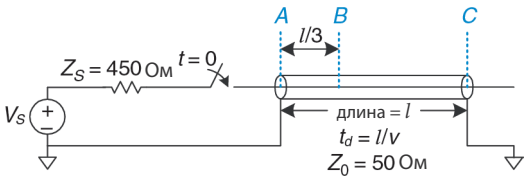


Рис. А.29 Линия передачи с несогласованным сопротивлением источника и нагрузки

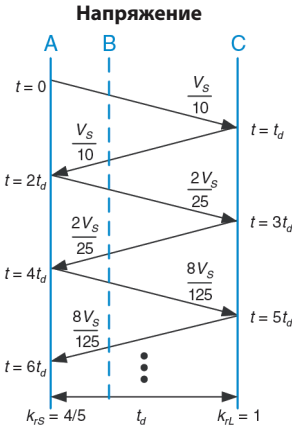


Рис. А.30 Диаграмма отражений для линии передачи на рис. А.29

Диаграмма отражений, представленная на **рис. А.30**, помогает представить отражения от обоих концов линии передачи. Горизонтальная ось представляет расстояние вдоль линии передачи, вертикальная ось представляет время, возрастающее сверху вниз. Две стороны диаграммы отражений представляют концы с источником и нагрузкой в линии передачи – точки A и C. Волны падающего и отраженного сигналов изображены диагональными линиями между точками A и C. В момент времени $t = 0$ сопротивление источника и линия передачи ведут себя как делитель напряжения, начиная распространять волну напряжением $V_S/10$ от точки A к точке C. В момент времени $t = t_d$ сигнал достигает точки C и полностью отражается ($k_{rL} = 1$). В момент времени $t = 2t_d$ отраженная волна напряжением $V_S/10$ достигает точки A и отражается с коэффициентом отражения $k_{rS} = 4/5$, порождая волну напряжением $2V_S/25$, распространяющуюся в сторону точки C, и т. д.

Напряжение в определенный момент времени в любой точке линии передачи равно сумме напряжений всех падающих и отраженных волн. Так, в момент времени $t = 1{,}1t_d$ напряжение в точке C равно $V_S/10 + V_S/10 = V_S/5$. В момент времени $t = 3{,}1t_d$ напряжение в точке C равно $V_S/10 + V_S/10 + 2V_S/25 + 2V_S/25 = 9V_S/25$ и т. д. На **рис. А.31** показан график зависимости напряжения от времени. Когда t стремится к бесконечности, достигается установившийся режим с напряжениями $V_A = V_B = V_C = V_S$.

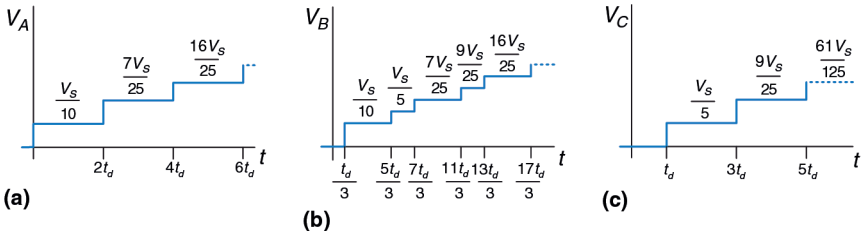


Рис. А.31 Формы напряжения и тока для линии на рис. А.29

А.8.5. Когда нужно применять модели линии передачи

Модели линии передачи для проводников необходимы всегда, когда задержка распространения сигнала в проводе t_d дольше, чем часть (например, 20 %) фронта сигнала (время нарастания и спада). Если задержка распространения сигнала в линии передачи меньше, то ее влияние на задержку распространения сигнала незначительно и отражения рассеиваются в процессе прохождения сигнала. Если задержка распространения сигнала в линии передачи больше, ее следует учитывать, чтобы точно предсказать задержку распространения и форму сигнала. В частности, отражения могут исказить цифровые характеристики формы сигнала, что приводит к неверным логическим операциям.

Следует помнить, что сигналы распространяются в печатной плате со скоростью примерно 15 см/нс. Для ТТЛ-логики, где фронты составляют 10 нс, проводники следует моделировать линиями передачи, только если они длиннее 30 см (10 нс × 15 см/нс × 20 %). Печатные проводники плат обычно короче 30 см, поэтому большинство печатных проводников можно моделировать как идеальные эквипотенциальные проводники. Напротив, многие современные микросхемы имеют фронты длительностью 2 нс или меньше, поэтому печатные проводники длиннее 6 см (около 2,5 дюйма) следует моделировать как линии передачи. Очевидно, что использование фронтов сигналов, которые короче, чем необходимо, ведет только к трудностям для разработчика.

У макетных плат нет плоскости заземления, поэтому электромагнитные поля сигналов неоднородны и трудно поддаются моделированию. Более того, эти поля взаимодействуют с другими сигналами. Это может привести к странным отражениям и взаимным помехам между сигналами. Таким образом, макетные платы ненадежны на частотах выше нескольких мегагерц.

С другой стороны, печатные платы имеют хорошие линии передачи с постоянными волновым сопротивлением и скоростью распространения по всей линии. Пока они нагружены сопротивлением источника или на-

грузки, согласованным с сопротивлением линии, проводники печатных плат не испытывают отражений.

А.8.6. Правильное подключение нагрузки к линии передачи

Существует два вида правильного подключения нагрузки к линии передачи, показанных на **рис. А.32**. При *параллельном подключении нагрузки* задающее устройство имеет малое сопротивление ($Z_S \approx 0$). Нагрузочный резистор Z_L с сопротивлением Z_0 располагается параллельно нагрузке (между входом принимающего устройства и общим проводом).

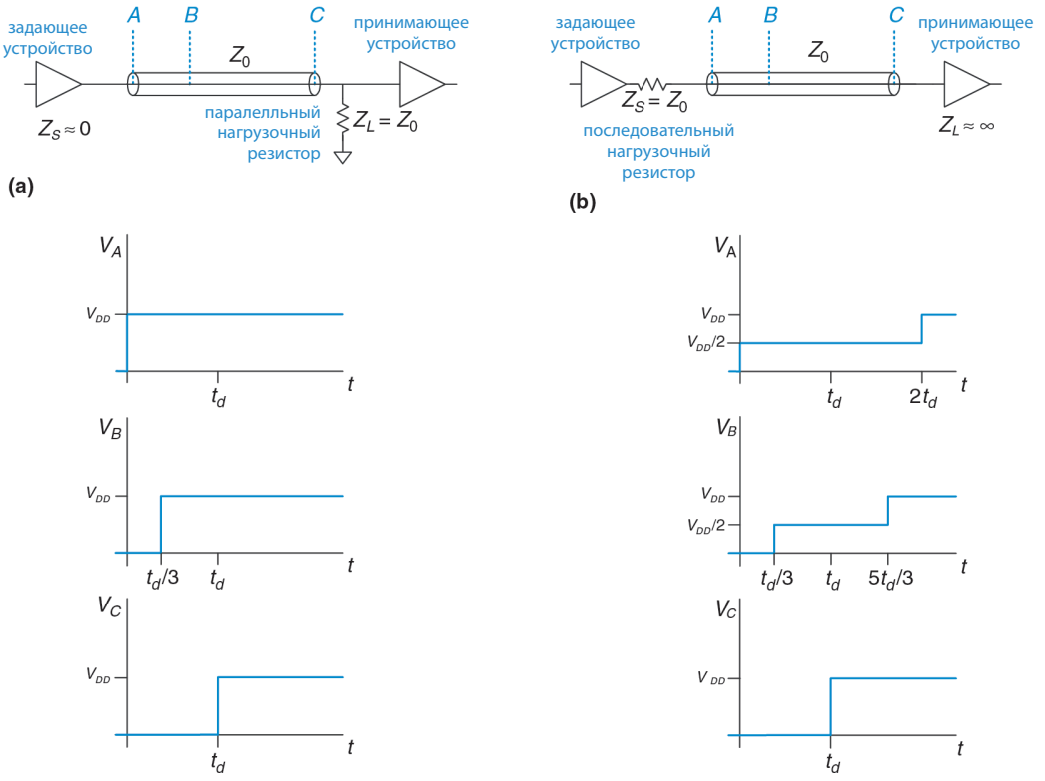


Рис. А.32 Варианты подключения нагрузки к линии передачи: (а) параллельная, (б) последовательная

Когда задающее устройство переключает напряжение от 0 до V_{DD}, оно начинает распространять волну с напряжением V_{DD} по линии. Волна поглощается согласованной нагрузкой, и отражений не происходит. При последовательном подключении нагрузки сопротивление Z_S располагается последовательно с задающим устройством, чтобы повысить сопротивле-

ние источника до Z_0. Нагрузка имеет высокое сопротивление ($Z_L \approx \infty$). Когда задающее устройство переключается, оно начинает распространять волну с напряжением $V_{DD}/2$ по линии. Волна отражается от открытого конца линии и возвращается, поднимая напряжение в линии до V_{DD}. Волна поглощается сопротивлением источника. Обе схемы похожи тем, что напряжение на принимающем устройстве изменяется от 0 до V_{DD} в момент времени $t = t_d$, именно так, как и требуется. Они различаются потребляемой мощностью и формой волны в каждой точке линии. Параллельная нагрузка постоянно рассеивает мощность на нагрузочном резисторе, когда линия находится под высоким напряжением. Последовательная нагрузка не рассеивает мощность постоянного тока, поскольку нагрузкой является разомкнутая цепь. При этом в линиях с последовательной нагрузкой точки в середине линии передачи изначально находятся под напряжением $V_{DD}/2$ до тех пор, пока их не достигнет отраженная волна. Если другие элементы будут подключены к середине линии, на них будет кратковременно действовать неправильный логический уровень. Поэтому последовательная нагрузка лучше работает в соединениях типа «точка–точка» с одним задающим и одним принимающим устройствами. Параллельная нагрузка лучше для шины со множеством принимающих устройств, поскольку принимающие устройства в середине линии никогда не подвергаются воздействию неправильных логических уровней.

A.8.7. Вывод формулы для Z_0

Z_0 равно отношению напряжения к току в волне, распространяющейся вдоль линии передачи. В данном разделе выводится формула для Z_0; вывод предполагает наличие некоторых знаний, полученных из анализа схем типа резистор–индуктивность–емкость (RLC).

Предположим, что ко входу полубесконечной линии передачи приложено ступенчатое напряжение (то есть отражений не происходит). На **рис. A.33** показана полубесконечная линия передачи и модель части этой линии длиной dx. R, L и C – *величины сопротивления, индуктивности и емкости на единицу длины*. На **рис. A.33 (b)** показана модель линии передачи с резистивным компонентом R. Такая модель называется моделью линии передачи с потерями, поскольку энергия рассеивается или теряется в сопротивлении проводника.

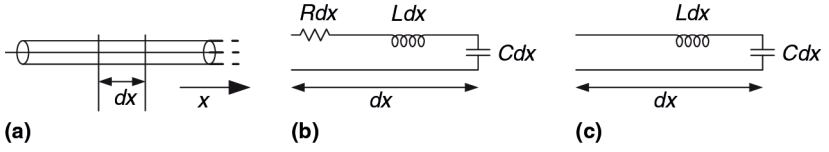


Рис. A.33 Модели линии передачи: (a) полубесконечный провод, (b) линия передачи с потерями, (c) идеальная линия передачи

Часто эти потери пренебрежимо малы, и можно упростить анализ, не принимая во внимание сопротивление и считая линию передачи идеальной, как показано на **рис. А.33 (с)**.

Напряжение и ток являются функциями времени и расстояния в линии передачи, как показано в уравнениях **(А.8)** и **(А.9)**:

$$\frac{\partial}{\partial x}V(x,t) = L\frac{\partial}{\partial t}I(x,t); \tag{A.8}$$

$$\frac{\partial}{\partial x}I(x,t) = C\frac{\partial}{\partial t}V(x,t). \tag{A.9}$$

Если взять производную по расстоянию от уравнения **(А.8)** и производную по времени от уравнения **(А.9)** и затем подставить одно в другое, то получим уравнение **(А.10)** – волновое уравнение:

$$\frac{\partial^2}{\partial x^2}V(x,t) = LC\frac{\partial^2}{\partial t^2}V(x,t). \tag{A.10}$$

Z_0 равно отношению напряжения к току в линии передачи, как показано на **рис. А.34 (а)**. Z_0 должно быть независимым от длины линии, так как поведение волны не может зависеть от удаленных объектов. Поскольку оно не зависит от длины, сопротивление должно быть равно Z_0 после удлинения линии на величину dx, как показано на **рис. А.34 (b)**.

Используя сопротивления индуктивности и емкости, можно представить соотношение на **рис. А.34** в виде уравнения:

$$Z_0 = j\omega L dx + [Z_0 \| (1/(j\omega C dx))]. \tag{A.11}$$

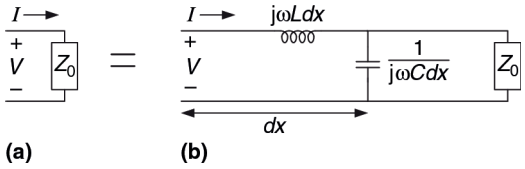


Рис. А.34 Модели линии передачи: (а) для всей линии, (b) с дополнительным отрезком длиной *dx*

После преобразований получим:

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 LC dx = 0. \tag{A.12}$$

В пределе при dx, стремящемся к 0, последнее слагаемое исчезает, и в итоге получается:

$$Z_0 = \sqrt{\frac{L}{C}}. \tag{A.13}$$

А.8.8. Вывод формулы для коэффициента отражения

Формула для коэффициента отражения выводится, используя принцип сохранения тока. На **рис. А.35** показана линия передачи с волновым сопротивлением Z_0 и сопротивлением нагрузки Z_L. Предположим, что падающая волна имеет напряжение V_i и ток I_i. Когда волна достигает нагрузки, некоторый ток I_L протекает через сопротивление нагрузки, вызывая падение напряжения V_L. Оставшийся ток отражается обратно в линию передачи в виде волны с напряжением V_r и током I_r. Z_0 равно отношению напряжения к току в волнах, распространяющихся вдоль линии, поэтому

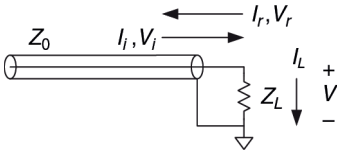


Рис. А.35 Линия передачи с падающими, отраженными и нагрузочными напряжениями и токами

$$\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0.$$

Напряжение в линии равно сумме напряжений падающей и отраженной волн. Ток, протекающий в линии передачи в положительном направлении, равен разности между токами падающей и отраженной волн:

$$V_L = V_i + V_r; \tag{A.14}$$

$$I_L = I_i - I_r. \tag{A.15}$$

Используя закон Ома и подставляя выражения для I_L, I_i и I_r в уравнение **(А.15)**, получим:

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0}. \tag{A.16}$$

После преобразований найдем решение для коэффициента отражения k_r:

$$\frac{V_i}{V_r} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r. \tag{A.17}$$

А.8.9. Линии передачи: подведение итогов

С помощью линий передачи можно смоделировать явление того, что сигналам необходимо время для распространения по длинным проводникам, поскольку скорость света конечна. Идеальная линия передачи име-

ет одинаковую величину индуктивности L и емкости на единицу длины и нулевое сопротивление. Линия передачи характеризуется волновым сопротивлением Z_0 и задержкой *распространения сигнала* t_d, которые можно вывести, зная индуктивность, емкость и длину проводника. Линии передачи имеют значительную временную задержку и шумовые эффекты при распространении сигналов, у которых длительность нарастания/спада меньше, чем примерно $5t_d$. Это означает, что для систем со временем нарастания/спада 2 нс печатные проводники плат длиннее 6 см должны рассматриваться как линии передачи, для того чтобы верно понимать их поведение.

Цифровая система, состоящая из элемента, управляющего длинным проводником, подключенным ко входу второго элемента, может быть смоделирована линией передачи, показанной на **рис. А.36**.

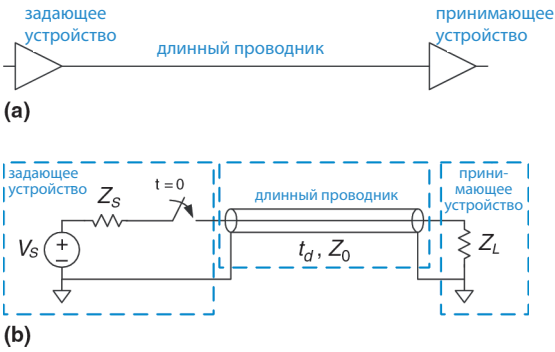


Рис. А.36 Цифровая система, смоделированная с помощью линии передачи

Источник напряжения, сопротивление источника Z_S и ключ имитируют первый элемент, переключающийся из 0 в 1 в момент времени 0. Элемент-источник не может поддерживать бесконечно большой ток, это моделируется при помощи Z_S. Z_S обычно мало для логических схем, но разработчик может добавить сопротивление, чтобы повысить Z_S и согласовать сопротивление линии. Вход второй схемы моделируется при помощи Z_L. У КМДП-схем обычно малый входной ток, поэтому величина Z_L может быть близка к бесконечности. Разработчик также может поставить параллельный резистор рядом со вторым элементом между его входом и общим проводом, чтобы Z_L согласовывало сопротивление линии.

Когда первый элемент переключается, волна напряжения начинает распространяться по линии передачи. Сопротивление источника и линия передачи формируют делитель напряжения, поэтому напряжение в падающей волне равно

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S}. \qquad \text{(A.18)}$$

В момент времени t_d волна достигает конца линии. Часть волны поглощается сопротивлением нагрузки, а часть отражается. Коэффициент отражения k_r показывает, какая часть волны отразилась. $k_r = V_r / V_i$, где V_r – напряжение отраженной волны, V_i – напряжение падающей волны.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0}. \tag{A.18}$$

Отраженная волна складывается с уже существующим в линии напряжением. Она достигает источника в момент времени $2t_d$, где часть поглощается, а часть снова отражается. Отражения продолжаются в обе стороны, и напряжение в линии в итоге достигает значения, которое было бы, если бы линия была простым эквипотенциальным проводником.

A.9. Экономика

Хотя разработка цифровых схем – забавное занятие и многие готовы заниматься этим бесплатно, большинство разработчиков и компаний стремятся заработать деньги. Поэтому экономические соображения – главный фактор при принятии проектных решений.

Стоимость цифровой системы может быть разделена на *единовременные инженерные издержки* (nonrecurring engineering costs, NRE) и *периодические издержки* (recurring costs). Единовременные издержки представляют собой стоимость разработки системы, которая включает в себя зарплату команды разработчиков, затраты на вычислительные мощности и программное обеспечение и стоимость производства первого рабочего образца. Полная стоимость разработчика в США в 2012 году (в том числе зарплата, страховка, пенсионное обеспечение и компьютер с необходимыми средствами разработки) составляла примерно $200 000 в год, так что затраты на разработку могут быть очень значительны. Периодические издержки – стоимость каждого дополнительного образца, которая включает в себя компоненты, производство, маркетинг, техническую поддержку и транспортировку.

Отпускная цена должна покрывать не только стоимость системы, но также и другие издержки вроде стоимости аренды помещений, налоги и зарплату персонала, которые непосредственно не задействованы в разработке (например, сторож и исполнительный директор). После всех этих издержек компании необходимо еще получить прибыль.

Пример A.3 БЕН ПЫТАЕТСЯ ЗАРАБОТАТЬ

Бен Битдидл разработал хитроумную схему для подсчета дождевых капель. Он решает продать прибор и попытаться заработать немного денег, но ему нужна помощь, чтобы решить, как реализовать схему. Он хочет использовать либо FPGA,

либо ASIC. Набор для разработки и тестирования на FPGA стоит $1500. Каждая FPGA стоит $17. ASIC стоит $600 000 за набор шаблонов и $4 за микросхему.

Независимо от того, какой метод исполнения он выберет, Бену необходимо установить корпусированную микросхему на печатную плату, которая будет стоить $1,50 за штуку. Он считает, что сможет продавать 1000 приборов в месяц. Бен заставил команду талантливых студентов разрабатывать схему в качестве выпускного проекта, поэтому разработка обошлась ему бесплатно.

Если продажная цена будет в два раза выше себестоимости (100%-ный предел прибыли) и жизненный цикл продукта 2 года, какое исполнение лучше выбрать?

Решение Бен выясняет полную стоимость каждой реализации за 2 года, расчеты приведены в **табл. А.4**. За 2 года Бен рассчитывает продать 24 000 устройств, общая стоимость дана в **табл. А.4** в каждой графе. Если жизненный цикл составляет всего 2 года, то реализация на FPGA очевидно побеждает. Стоимость одного изделия равна $445 500 / 24 000 = $18,56, и продажная цена составляет $37,13 за единицу товара, чтобы получить 100%-ную прибыль. Вариант на ASIC стоил бы $732 000 / 24 000 = $30,50 и продавался бы за $61 за изделие.

Таблица А.4 Сравнение стоимости ASIC и FPGA

Стоимость	ASIC	FPGA
Единовременные расходы	$600 000	$1500
микросхема	$4	$17
печатная плата	$1,50	$1,50
ВСЕГО	$600 000 + (24 000 × $5,50) = $732 000	$1500 + (24 000 × $18,50) = $445 500
за единицу	$30,50	$18,56

Пример А.4 БЕН СТАНОВИТСЯ ЖАДНЫМ

После того как Бен увидел рекламу своего товара, он решает, что сможет продавать больше микросхем в месяц, чем первоначально планировалось. Если бы он выбрал реализацию на ASIC, сколько приборов в месяц ему бы надо было продать, чтобы эта реализация стала более прибыльной, чем FPGA?

Решение Бен определяет минимальное количество изделий N, которое ему необходимо продать за 2 года:

$$\$600\,000 + (N \times \$5{,}50) = \$1500 + (N \times \$18{,}50).$$

Решение уравнения дает N = 46 039 устройств, или 1919 устройств в месяц. Ему придется почти удвоить ежемесячные продажи, чтобы получить прибыть от варианта с ASIC.

Пример А.5 БЕН СТАНОВИТСЯ МЕНЕЕ ЖАДНЫМ

Бен понимает, что он слишком многого хочет, и не думает, что сможет продать больше 1000 устройств в месяц. Но он считает, что жизненный цикл может быть

больше 2 лет. При объеме продаж 1000 устройств в месяц каков должен быть жизненный цикл, чтобы сделать ASIC-реализацию прибыльной?

Решение Если Бен продаст больше 46 039 устройств, то ASIC-реализация будет лучшим выбором. Так что Бену придется продавать по 1000 изделий в течение как минимум 47 месяцев (примерно), что составляет почти 4 года. К тому времени устройство, вероятно, станет устаревшим.

Микросхемы обычно приобретаются у дистрибьютора, а не напрямую у производителя (если не заказываются десятки тысяч микросхем). Digikey (www.digikey.com) – основной дистрибьютор, продающий большую номенклатуру изделий электроники. Jameco (www.jameco.com) и All Electronics (www.allelectronics.com) имеют разнообразные каталоги с конкурентными ценами и хорошо подходят для любителей.

ПРИЛОЖЕНИЕ B

Система команд RISC-V

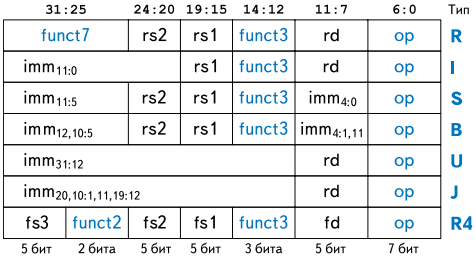


Рис. B1. Формат команд RISC-V

- `imm`: Константа со знаком в $imm_{11:0}$
- `uimm`: 5-битная константа без знака в $imm_{4:0}$
- `upimm`: 20 старших битов 32-битной константы в $imm_{31:12}$
- `Address`: Адрес памяти: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: Данные по адресу Address
- `BTA`: Адрес условного перехода: PC + SignExt({$imm_{12:1}$, 1'b0})
- `JTA`: Адрес безусловного перехода: PC + SignExt({imm20:1,1'b0})
- `label`: Текстовый указатель адреса перехода
- `SignExt`: Значение, дополненное знаком до 32 бит
- `ZeroExt`: Значение, дополненное знаком до 32 бит
- `csr`: Регистр управления и состояния

Таблица B1 Целочисленные инструкции RISC-V (RV32I)

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte загрузка байта	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half загрузка половины слова	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word загрузка слова	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned загрузка байта без знака	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned загрузка половины слова без знака	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate сложение с константой	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate логический сдвиг константы влево	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate установить регистр назначения в 1, если регистр-источник меньше, чем константа	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned установить регистр назначения в 1, если регистр-источник меньше, чем константа. Операция беззнаковая	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate ИСКЛЮЧАЮЩЕЕ ИЛИ с константой	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate логический сдвиг константы вправо	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm. арифметический сдвиг константы вправо	`rd = rs1 >>> uimm`

Таблица B1 (окончание)

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate логическая операция ИЛИ с константой	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate логическая операция И с константой	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC прибавить старшую половину константы к счетчику команд	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte сохранить байт в памяти	$[Address]_{7:0} = rs2_{7:0}$
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half сохранить половину слова в памяти	$[Address]_{15:0} = rs2_{15:0}$
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word сохранить слово в памяти	$[Address]_{31:0} = rs2$
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add сложение	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub вычитание	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical логический сдвиг влево	$rd = rs1 << rs2_{4:0}$
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than установить rd в 1, если rs1 меньше, чем rs2	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned установить rd в 1, если rs1 меньше, чем rs2, беззнаковая операция	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical логический сдвиг вправо	$rd = rs1 >> rs2_{4:0}$
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic арифметический сдвиг вправо	$rd = rs1 >>> rs2_{4:0}$
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or логическая операция ИЛИ	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and логическая операция И	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate загрузить старшую половину константы в регистр	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if = переход, если равно	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠ переход, если не равно	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if < переход, если меньше	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥ переход, если больше или равно	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned переход, если меньше, без учета знака	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned переход, если больше, без учета знака	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register переход с возвратом по адресу в регистре	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link переход с возвратом	`PC = JTA, rd = PC + 4`

* Закодировано в старших семи битах поля константы ($imm_{31:25}$).

Таблица B2 Дополнительные целочисленные инструкции RV64I

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0000011 (3)	011	–	I	`ld rd, imm(rs1)`	load double word загрузка двойного слова	`rd = [Address]63:0`
0000011 (3)	110	–	I	`lwu rd, imm(rs1)`	load word unsigned загрузка двойного слова без знака	`rd = ZeroExt([Address]31:0)`
0011011 (27)	000	–	I	`addiw rd, rs1, imm`	add immediate word сложение слова с константой	`rd = SignExt((rs1 + SignExt(imm))31:0)`
0011011 (27)	001	0000000	I	`slliw rd, rs1, uimm`	shift left logical immediate word логический сдвиг влево слова на заданное в константе количество битов	`rd = SignExt((rs131:0 << uimm)31:0)`
0011011 (27)	101	0000000	I	`srliw rd, rs1, uimm`	shift right logical immediate word логический сдвиг вправо слова на заданное в константе количество битов	`rd = SignExt((rs131:0 >> uimm)31:0)`
0011011 (27)	101	0100000	I	`sraiw rd, rs1, uimm`	shift right arith. immediate word арифметический сдвиг вправо слова на заданное в константе количество битов	`rd = SignExt((rs131:0 >>> uimm)31:0)`
0100011 (35)	011	–	S	`sd rs2, imm(rs1)`	store double word сохранить двойное слово в памяти	`[Address]63:0 = rs2`
0111011 (59)	000	0000000	R	`addw rd, rs1, rs2`	add word сложение двух регистров, содержащих 32-битные слова	`rd = SignExt((rs1 + rs2)31:0)`
0111011 (59)	000	0100000	R	`subw rd, rs1, rs2`	subtract word вычитание двух регистров, содержащих 32-битные слова	`rd = SignExt((rs1 – rs2)31:0)`
0111011 (59)	001	0000000	R	`sllw rd, rs1, rs2`	shift left logical word логический сдвиг слова влево	`rd = SignExt((rs131:0 << rs24:0)31:0)`
0111011 (59)	101	0000000	R	`srlw rd, rs1, rs2`	shift right logical word логический сдвиг слова вправо	`rd = SignExt((rs131:0 >> rs24:0)31:0)`
0111011 (59)	101	0100000	R	`sraw rd, rs1, rs2`	shift right arithmetic word арифметический сдвиг слова вправо	`rd = SignExt((rs131:0 >>> rs24:0)31:0)`

В наборе инструкций RV64I регистры 64-битные, но инструкции остаются 32-битными. Термин «word» (слово) обычно означает 32-битное значение. В наборе RV64I инструкции сдвига с константой используют 6-битную константу $uimm_{5:0}$, но для сдвига слова старший значащий бит значения сдвига ($uimm_5$) должен быть равен нулю. Инструкции, которые заканчиваются символом «w» (word, слово), работают с младшей половиной 64-битных регистров. 64-битный результат получается за счет дополнения знаковым или нулевым битом.

Таблица B3 RVF/D: инструкции RISC-V одинарной и двойной точности с плавающей запятой

op	funct3	funct7	rs2	Тип	Инструкция	Описание	Операция
1000011 (67)	rm	fs3, fmt	–	R4	`fmadd fd,fs1,fs2,fs3`	multiply-add умножение-сложение	`fd = fs1 * fs2 + fs3`
1000111 (71)	rm	fs3, fmt	–	R4	`fmsub fd,fs1,fs2,fs3`	multiply-subtract умножение-вычитание	`fd = fs1 * fs2 – fs3`
1001011 (75)	rm	fs3, fmt	–	R4	`fnmsub fd,fs1,fs2,fs3`	negate multiply-add умножение-сложение с инверсией	`fd = –(fs1 * fs2 + fs3)`
1001111 (79)	rm	fs3, fmt	–	R4	`fnmadd fd,fs1,fs2,fs3`	negate multiply-subtract умножение-вычитание с инверсией	`fd = –(fs1 * fs2 - fs3)`

Таблица B3 (продолжение)

op	funct3	funct7	rs2	Тип	Инструкция	Описание	Операция
1010011 (83)	rm	00000, fmt	–	R	`fadd fd,fs1,fs2`	add сложение	`fd = fs1 + fs2`
1010011 (83)	rm	00001, fmt	–	R	`fsub fd,fs1,fs2`	subtract вычитание	`fd = fs1 – fs2`
1010011 (83)	rm	00010, fmt	–	R	`fmul fd,fs1,fs2`	multiply умножение	`fd = fs1 * fs2`
1010011 (83)	rm	00011, fmt	–	R	`fdiv fd,fs1,fs2`	divide деление	`fd = fs1 / fs2`
1010011 (83)	rm	01011, fmt	00000	R	`fsqrt fd,fs1`	square root квадратный корень	`fd = sqrt(fs1)`
1010011 (83)	000	00100, fmt	–	R	`fsgnj fd,fs1,fs2`	sign injection в fd помещается значение fs1, кроме знака, знак берется от fs2	`fd = fs1, sign = sign(fs2)`
1010011 (83)	001	00100, fmt	–	R	`fsgnjn fd,fs1,fs2`	negate sign injection в fd помещается значение fs1, кроме знака, знак берется как инверсия знака fs2	`fd = fs1, sign = –sign(fs2)`
1010011 (83)	010	00100, fmt	–	R	`fsgnjx fd,fs1,fs2`	xor sign injection в fd помещается значение fs1, кроме знака, знак берется как логическая операция И между знаками fs1 и fs2	`fd = fs1, sign = sign(fs2) ^ sign(fs1)`
1010011 (83)	000	00101, fmt	–	R	`fmin fd,fs1,fs2`	min выбрать наименьший операнд	`fd = min(fs1, fs2)`
1010011 (83)	001	00101, fmt	–	R	`fmax fd,fs1,fs2`	max выбрать наибольший операнд	`fd = max(fs1, fs2)`
1010011 (83)	010	10100, fmt	–	R	`feq rd,fs1,fs2`	compare = в rd поместить 1, если fs1 = fs2	`rd = (fs1 == fs2)`
1010011 (83)	001	10100, fmt	–	R	`flt rd,fs1,fs2`	compare < в rd поместить 1, если fs1 < fs2	`rd = (fs1 < fs2)`
1010011 (83)	000	10100, fmt	–	R	`fle rd,fs1,fs2`	compare ≤ в rd поместить 1, если fs1 ≤ fs2	`rd = (fs1 ≤ fs2)`
1010011 (83)	001	11100, fmt	00000	R	`fclass rd,fs1`	classify в rd помещается число, описывающее тип значения, которое хранится в fs1 (например, 0, если fs1 = –∞, или 3, если fs1 = –0, и т. д.)	`rd = classification of fs1`
Только RVF							
0000111 (7)	010	–	–	I	`flw fd, imm(rs1)`	load float чтение значения с плавающей запятой из памяти	`fd = [Address]31:0`
0100111 (39)	010	–	–	S	`fsw fs2,imm(rs1)`	store float сохранение значения с плавающей запятой в памяти	`[Address]31:0 = fd`
1010011 (83)	rm	1100000	00000	R	`fcvt.w.s rd, fs1`	convert to integer преобразовать в целое число	`rd = integer(fs1)`
1010011 (83)	rm	1100000	00001	R	`fcvt.wu.s rd, fs1`	convert to unsigned integer преобразовать в целое число без знака	`rd = unsigned(fs1)`

Таблица B1 (окончание)

op	funct3	funct7	rs2	Тип	Инструкция		Описание	Операция
1010011 (83)	rm	1101000	00000	R	`fcvt.s.w`	`fd, rs1`	convert int to float конвертировать целое число в формат с плавающей запятой	`fd = float(rs1)`
1010011 (83)	rm	1101000	00001	R	`fcvt.s.wu`	`fd, rs1`	convert unsigned to float конвертировать целое число в формат с плавающей запятой без знака	`fd = float(rs1)`
1010011 (83)	000	1110000	00000	R	`fmv.x.w`	`rd, fs1`	move to integer register загрузить в целочисленный регистр	`rd = fs1`
1010011 (83)	000	1111000	00000	R	`fmv.w.x`	`fd, rs1`	move to f.p. register загрузить в регистр с плавающей запятой	`fd = rs1`
Только RVD								
0000111 (7)	011	-	–	I	`fld`	`fd, imm(rs1)`	load double загрузить число с двойной точностью	`fd = [Address]63:0`
0100111 (39)	011	–	–	S	`fsd`	`fs2,imm(rs1)`	store double сохранить число с двойной точностью	`[Address]63:0 = fd`
1010011 (83)	rm	1100001	00000	R	`fcvt.w.d`	`rd, fs1`	convert to integer преобразовать в целое число	`rd = integer(fs1)`
1010011 (83)	rm	1100001	00001	R	`fcvt.wu.d`	`rd, fs1`	convert to unsigned integer преобразовать в целое число без знака	`rd = unsigned(fs1)`
1010011 (83)	rm	1101001	00000	R	`fcvt.d.w`	`fd, rs1`	convert int to double конвертировать целое число в формат с двойной точностью	`fd = double(rs1)`
1010011 (83)	rm	1101001	00001	R	`fcvt.d.wu`	`fd, rs1`	convert unsigned to double конвертировать целое число без знака в формат с двойной точностью	`fd = double(rs1)`
1010011 (83)	rm	0100000	00001	R	`fcvt.s.d`	`fd, fs1`	convert double to float преобразовать из формата с двойной точностью в формат с плавающей запятой	`fd = float(fs1)`
1010011 (83)	rm	0100001	00000	R	`fcvt.d.s`	`fd, fs1`	convert float to double преобразовать из формата с плавающей запятой в формат с двойной точностью	`fd = double(fs1)`

fs1, **fs2**, **fs3**, **fd**: регистры с плавающей запятой. **fs1**, **fs2** и **fd** закодированы в полях **rs1**, **rs2** и **rd**; только тип R4 использует еще и поле **fs3**. **fmt**: точность вычислений (одинарная = 00_2, двойная = 01_2, четверная = 11_2). **rm**: режим округления: 0 = до ближайшего, 1 = к нулю, 2 = вниз, 3 = вверх, 4 = до ближайшего (с наибольшим значением), 7 = динамический. **sign(fs1)** – знак **fs1**.

Таблица B4 Номера и имена регистров

Имя	Номер регистра	Назначение
zero	x0	Фиксированное нулевое значение
ra	x1	Адрес возврата
sp	x2	Указатель стека
gp	x3	Глобальный указатель
tp	x4	Указатель потока
t0–2	x5–7	Рабочие (временные) регистры
s0/fp	x8	Сохраняемый регистр / Указатель кадра
s1	x9	Сохраняемый регистр
a0–1	x10–11	Аргументы функции / Возвращаемые значения
a2–7	x12–17	Аргументы функции
s2–11	x18–27	Сохраняемые регистры
t3–6	x28–31	Временные (рабочие) регистры

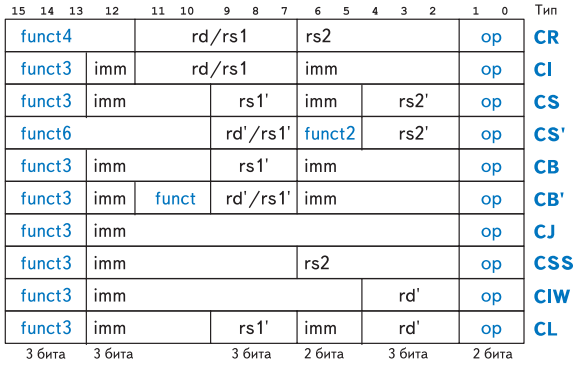


Рис. B2 Формат компактной инструкции RISC-V

Таблица B5 Инструкции умножения и деления RISC-V (RVM)

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply умножение	`rd = (rs1 * rs2)31:0`
0110011 (51)	001	0000001	R	`mulh rd, rs1, rs2`	multiply high signed signed умножение старших разрядов со знаком на число со знаком	`rd = (rs1 * rs2)63:32`
0110011 (51)	010	0000001	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned умножение старших разрядов со знаком на число без знака	`rd = (rs1 * rs2)63:32`
0110011 (51)	011	0000001	R	`mulhu rd, rs1, rs2`	multiply high unsigned unsigned умножение старших разрядов без знака на число без знака	`rd = (rs1 * rs2)63:32`
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed) деление (со знаком)	`rd = rs1 / rs2`
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned деление (без знака)	`rd = rs1 / rs2`
0110011 (51)	110	0000001	R	`rem rd, rs1, rs2`	remainder (signed) остаток от деления (со знаком)	`rd = rs1 % rs2`
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned остаток от деления (без знака)	`rd = rs1 % rs2`

Таблица B6 Компактные инструкции RISC-V (RVC)

op	$instr_{15:10}$	funct2	Тип	Компактные инструкция	32-битный эквивалент
00 (0)	000–––	–	CIW	`c.addi4spn rd', sp, imm`	`addi rd', sp, ZeroExt(imm)*4`
00 (0)	001–––	–	CL	`c.fld fd', imm(rs1')`	`fld fd', (ZeroExt(imm)*8)(rs1')`
00 (0)	010–––	–	CL	`c.lw rd', imm(rs1')`	`lw rd', (ZeroExt(imm)*4)(rs1')`
00 (0)	011–––	–	CL	`c.flw fd', imm(rs1')`	`flw fd', (ZeroExt(imm)*4)(rs1')`
00 (0)	101–––	–	CS	`c.fsd fs2', imm(rs1')`	`fsd fs2', (ZeroExt(imm)*8)(rs1')`
00 (0)	110–––	–	CS	`c.sw rs2', imm(rs1')`	`sw rs2', (ZeroExt(imm)*4)(rs1')`
00 (0)	111–––	–	CS	`c.fsw fs2', imm(rs1')`	`fsw fs2', (ZeroExt(imm)*4)(rs1')`
01 (1)	000000	–	CI	`c.nop (rs1=0,imm=0)`	`nop`
01 (1)	000–––	–	CI	`c.addi rd, imm`	`addi rd, rd, SignExt(imm)`

Таблица B6 (окончание)

op	$instr_{15:10}$	funct2	Тип	Компактные инструкция	32-битный эквивалент
01 (1)	001–––	–	CJ	`c.jal label`	`jal ra, label`
01 (1)	010–––	–	CI	`c.li rd, imm`	`addi rd, x0, SignExt(imm)`
01 (1)	011–––	–	CI	`c.lui rd, imm`	`lui rd, {14{imm5}, imm}`
01 (1)	011–––	–	CI	`c.addi16sp sp, imm`	`addi sp, sp, SignExt(imm)*16`
01 (1)	100–00	–	CB'	`c.srli rd', imm`	`srli rd', rd', imm`
01 (1)	100–01	–	CB'	`c.srai rd', imm`	`srai rd', rd', imm`
01 (1)	100–10	–	CB'	`c.andi rd', imm`	`andi rd', rd', SignExt(imm)`
01 (1)	100011	00	CS'	`c.sub rd', rs2'`	`sub rd', rd', rs2'`
01 (1)	100011	01	CS'	`c.xor rd', rs2'`	`xor rd', rd', rs2'`
01 (1)	100011	10	CS'	`c.or rd', rs2'`	`or rd', rd', rs2'`
01 (1)	100011	11	CS'	`c.and rd', rs2'`	`and rd', rd', rs2'`
01 (1)	101–––	–	CJ	`c.j label`	`jal x0, label`
01 (1)	110–––	–	CB	`c.beqz rs1', label`	`beq rs1', x0, label`
01 (1)	111–––	–	CB	`c.bnez rs1', label`	`bne rs1', x0, label`
10 (2)	000–––	–	CI	`c.slli rd, imm`	`slli rd, rd, imm`
10 (2)	001–––	–	CI	`c.fldsp fd, imm`	`fld fd, (ZeroExt(imm)*8)(sp)`
10 (2)	010–––	–	CI	`c.lwsp rd, imm`	`lw rd, (ZeroExt(imm)*4)(sp)`
10 (2)	011–––	–	CI	`c.flwsp fd, imm`	`flw fd, (ZeroExt(imm)*4)(sp)`
10 (2)	1000––	–	CR	`c.jr rs1 (rs1≠0,rs2=0)`	`jalr x0, rs1, 0`
10 (2)	1000––	–	CR	`c.mv rd, rs2 (rd ≠0,rs2≠0)`	`add rd, x0, rs2`
10 (2)	1001––	–	CR	`c.ebreak (rs1=0,rs2=0)`	`ebreak`
10 (2)	1001––	–	CR	`c.jalr rs1 (rs1≠0,rs2=0)`	`jalr ra, rs1, 0`
10 (2)	1001––	–	CR	`c.add rd, rs2 (rs1≠0,rs2≠0)`	`add rd, rd, rs2`
10 (2)	101–––	–	CSS	`c.fsdsp fs2, imm`	`fsd fs2, (ZeroExt(imm)*8)(sp)`
10 (2)	110–––	–	CSS	`c.swsp rs2, imm`	`sw rs2, (ZeroExt(imm)*4)(sp)`
10 (2)	111–––	–	CSS	`c.fswsp fs2, imm`	`fsw fs2, (ZeroExt(imm)*4)(sp)`

rs1', **rs2'**, **rd'**: 3-битный указатель на регистры 8–15: 000_2 = x8 или f8, 001_2 = x9 или f9 и т. д.

Таблица B7 Псевдоинструкции RISC-V

Псевдоинструкция	Инструкция RISC-V	Описание	Операция
`nop`	`addi x0, x0, 0`	no operation «пустая» операция	
`li rd, imm11:0`	`addi rd, x0, imm11:0`	load 12-bit immediate загрузить 12-битную константу	`rd = SignExtend(imm11:0)`
`li rd, imm31:0`	`lui rd, imm31:12*` `addi rd, rd, imm11:0`	load 32-bit immediate загрузить 32-битную константу	`rd = imm31:0`
`mv rd, rs1`	`addi rd, rs1, 0`	move (also called "register copy") скопировать значение в другой регистр	`rd = rs1`
`not rd, rs1`	`xori rd, rs1, −1`	one's complement преобразовать в обратный код	`rd = ~rs1`
`neg rd, rs1`	`sub rd, x0, rs1`	two's complement преобразовать в дополнительный код	`rd = −rs1`
`seqz rd, rs1`	`sltiu rd, rs1, 1`	set if = 0 проверка на равенство нулю	`rd = (rs1 == 0)`
`snez rd, rs1`	`sltu rd, x0, rs1`	set if ≠ 0 проверка на неравенство нулю	`rd = (rs1 ≠ 0)`
`sltz rd, rs1`	`slt rd, rs1, x0`	set if < 0 проверка на то, что регистр меньше нуля	`rd = (rs1 < 0)`

Таблица B7 (окончание)

Псевдоинструкция	Инструкция RISC-V	Описание	Операция
`sgtz rd, rs1`	`slt rd, x0, rs1`	set if > 0 проверка на то, что регистр больше нуля	`rd = (rs1 > 0)`
`beqz rs1, label`	`beq rs1, x0, label`	branch if = 0 переход, если равно нулю	`if (rs1 == 0) C = label`
`bnez rs1, label`	`bne rs1, x0, label`	branch if ≠ 0 переход, если не равно нулю	`if (rs1 ≠ 0) PC = label`
`blez rs1, label`	`bge x0, rs1, label`	branch if ≤ 0 переход, если меньше или равно нулю	`if (rs1 ≤ 0) PC = label`
`bgez rs1, label`	`bge rs1, x0, label`	branch if ≥ 0 переход, если больше или равно нулю	`if (rs1 ≥ 0) PC = label`
`bltz rs1, label`	`blt rs1, x0, label`	branch if < 0 переход, если меньше	`if (rs1 < 0) PC = label`
`bgtz rs1, label`	`blt x0, rs1, label`	branch if > 0 переход, если больше	`if (rs1 > 0) PC = label`
`ble rs1, rs2, label`	`bge rs2, rs1, label`	branch if ≤ переход, если меньше или равно	`if (rs1 ≤ rs2) PC = label`
`bgt rs1, rs2, label`	`blt rs2, rs1, label`	branch if > переход, если больше	`if (rs1 > rs2) PC = label`
`bleu rs1, rs2, label`	`bgeu rs2, rs1, label`	branch if ≤ (unsigned) переход, если меньше или равно (без знака)	`if (rs1 ≤ rs2) PC = label`
`bgtu rs1, rs2, label`	`bltu rs2, rs1, offset`	branch if > (unsigned) переход, если больше (без знака)	`if (rs1 > rs2) PC = label`
`j label`	`jal x0, label`	jump безусловный переход	`PC = label`
`jal label`	`jal ra, label`	jump and link безусловный переход с возвратом	`PC = label, ra = PC + 4`
`jr rs1`	`jalr x0, rs1, 0`	jump register безусловный переход по адресу из регистра	`PC = rs1`
`jalr rs1`	`jalr ra, rs1, 0`	jump and link register безусловный переход по адресу из регистра с возвратом	`PC = rs1, ra = PC + 4`
`ret`	`jalr x0, ra, 0`	return from function возврат из функции	`PC = ra`
`call label`	`jal ra, label`	call nearby function вызов функции, расположенной по ближнему адресу	`PC = label, ra = PC + 4`
`call label`	`auipc ra, offset`$_{31:12}$* `jalr ra, ra, offset`$_{11:0}$	call far away function вызов функции, расположенной по дальнему адресу	`PC = PC + offset, ra = PC + 4`
`la rd, symbol`	`auipc rd, symbol`$_{31:12}$* `addi rd, rd, symbol`$_{11:0}$	load address of global variable загрузить адрес глобальной переменной	`rd = PC + symbol`
`l{b|h|w} rd, symbol`	`auipc rd, symbol`$_{31:12}$* `l{b|h|w} rd, symbol`$_{11:0}$`(rd)`	load global variable загрузить глобальную переменную	`rd = [PC + symbol]`
`s{b|h|w} rs2, symbol, rs1`	`auipc rs1, symbol`$_{31:12}$* `s{b|h|w} rs2, symbol`$_{11:0}$`(rs1)`	store global variable сохранить глобальную переменную	`[PC + symbol] = rs2`
`csrr rd, csr`	`csrrs rd, csr, x0`	read CSR прочитать CSR	`rd = csr`
`csrw csr, rs1`	`csrrw x0, csr, rs1`	write CSR сохранить CSR	`csr = rs1`

* Если 11 бит константы / смещения / символа равен 1, старшая половина константы увеличивается на 1. symbol и offset – 32-битные адреса относительно счетчика команд для метки и глобальной переменной соответственно.

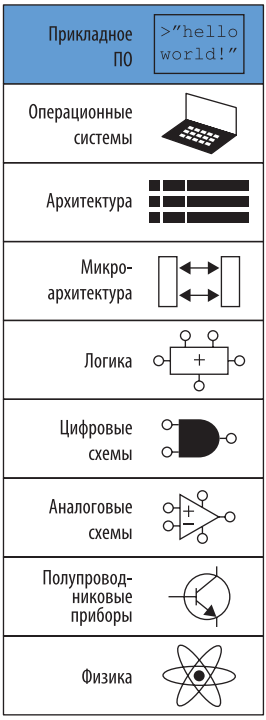
Таблица B8 Привилегированные и CSR-инструкции

op	funct3	Тип	Инструкция	Описание		Операция
1110011 (115)	000	I	`ecall`	transfer control to OS передать управление ОС	(imm=0)	
1110011 (115)	000	I	`ebreak`	transfer control to debugger передать управление отладчику	(imm=1)	
1110011 (115)	000	I	`uret`	return from user exception возврат из обработчика пользовательского прерывания	(rs1=0, rd=0,imm=2)	`PC = uepc`
1110011 (115)	000	I	`sret`	return from supervisor exception возврат из обработчика прерывания от супервизора	(rs1=0, rd=0,imm=258)	`PC = sepc`
1110011 (115)	000	I	`mret`	return from machine exception возврат из обработчика внутреннего прерывания	(rs1=0, rd=0,imm=770)	`PC = mepc`
1110011 (115)	001	I	`csrrw rd,csr,rs1`	CSR read/write запись/чтение CSR	(imm=CSR number)	`rd = csr, csr = rs1`
1110011 (115)	010	I	`csrrs rd,csr,rs1`	CSR read/set чтение/установка CSR	(imm=CSR number)	`rd = csr, csr = csr | rs1`
1110011 (115)	011	I	`csrrc rd,csr,rs1`	CSR read/clear чтение/очистка CSR	(imm=CSR number)	`rd = csr, sr = csr & ~rs1`
1110011 (115)	101	I	`csrrwi rd,csr,uimm`	CSR read/write immediate чтение/запись CSR с константой	(imm=CSR number)	`rd = csr, csr = ZeroExt(uimm)`
1110011 (115)	110	I	`csrrsi rd,csr,uimm`	CSR read/set immediate чтение/установка CSR с константой	(imm=CSR number)	`rd = csr,` `csr = csr | ZeroExt(uimm)`
1110011 (115)	111	I	`csrrci rd,csr,uimm`	CSR read/clear immediate чтение/очистка CSR с константой	(imm=CSR number)	`rd = csr,` `csr = csr & ~ZeroExt(uimm)`

В привилегированных и CSR-инструкциях 5-битная константа без знака `uimm` закодирована в поле **rs1**.

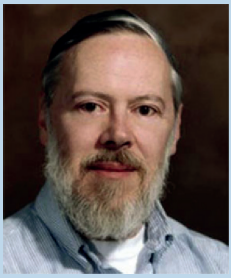
Приложение C

Программирование на языке C

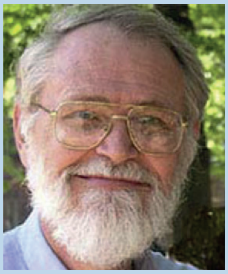


C.1. Введение

Цель этой книги – показать работу компьютера на различных уровнях, начиная с транзисторных схем и заканчивая выполнением программ. Первые пять глав книги описывают нижние уровни абстракции от транзисторов к логическим вентилям и далее к логическим схемам. В шестой, седьмой и восьмой главах сначала рассматривается архитектура, чтобы потом опуститься на уровень микроархитектуры и перекинуть мостик от аппаратного к программному обеспечению. **Приложение C** могло бы располагаться между **главами 5** и **6**. Оно представляет собой краткое введение в язык программирования C, являющийся наивысшим уровнем



Деннис Ритчи, 1941–2011



Брайан Керниган, 1942–

Первым официальным описанием языка C стала классическая книга Брайана Кернигана и Денниса Ритчи «Язык программирования C» (The C Programming Language), которая была опубликована в 1978 году. В 1989 году Американский национальный институт стандартов (ANSI) расширил и стандартизировал спецификации языка C. Эта версия языка стала известна под названиями ANSI C, Standard C или C89. Вскоре после этого, в 1990 году, стандарт ANSI C был принят Международной организацией по стандартизации (ISO) и Международной электротехнической комиссией (IEC). В 1999 году организации ISO/IEC пересмотрели стандарт и опубликовали обновленную версию спецификаций языка, получившую название C99, которую мы и будем рассматривать в этой книге*.

* В 2011 году организации ISO/IEC выпустили обновленную версию стандарта, получившую название C11, в 2014 году – версию C14, а в 2017 году – версию C17. – *Прим. перев.*

абстракции в этой книге. Такой порядок изложения связывает описание аппаратной архитектуры с практикой программирования, где у читателя, возможно, уже имеется личный опыт. Данный материал вынесен в отдельное приложение, чтобы читатель имел возможность либо изучить материал, либо пропустить его, в зависимости от степени знакомства с предметом.

Чтобы описать действия, которые должен выполнить компьютер, используются различные языки программирования. По существу, компьютер выполняет команды на машинном языке, которые состоят из единиц и нулей, как было описано в **главе 6**. Но создание программ напрямую в машинных кодах – это утомительное и медленное занятие, что побуждает использовать языки более высокого уровня для повышения продуктивности программистов. В **табл. C.1** перечислены некоторые языки программирования различных уровней абстракции.

Таблица C.1 Языки программирования в порядке убывания уровня абстракции

Язык программирования	Описание
Matlab	Богатые возможности для записи математических операций
Perl	Небольшие программы для обработки текстов
Python	Внимание к повышению читаемости текста программы
Java	Безопасное выполнение на различных платформах
C	Гибкий доступ к широкому спектру системных ресурсов, включая драйверы устройств
Ассемблер	Представление машинного кода в удобной для чтения форме
Машинные коды	Двоичное представление программы

Один из самых популярных языков программирования – это **язык C**, который был создан в период между 1969 и 1973 годами группой сотрудников Bell Laboratories, включавшей Денниса Ритчи и Брайана Кернигана. Целью группы было создание языка высокого уровня, позволяющего переписать код операционной системы UNIX, первоначально написанной на языке ассемблера. По многочисленным оценкам, язык C (ставший базовым для семейства таких языков, как C++, C# и Objective-C) является самым популярным языком программирования

из ныне существующих. Его широкое признание вызвано удачным сочетанием ряда свойств языка. Вот некоторые из них:

- доступность на большом количестве платформ от микроконтроллеров до суперкомпьютеров;
- относительная простота и большое количество пользователей;
- средний уровень абстракции языка, что позволяет писать программы продуктивнее по сравнению с использованием языка ассемблера, а с другой стороны дает программисту представление, как будет выполняться код программы;
- пригодность для создания высокопроизводительных программ;
- возможность напрямую работать с аппаратным обеспечением.

Это приложение посвящено языку C по многим причинам, самой важной из которых является возможность напрямую обращаться к оперативной памяти. Данное свойство языка хорошо иллюстрирует связь между аппаратным и программным обеспечением, на которую мы обращаем внимание в этой книге. Все инженеры и ученые, работающие в области разработки аппаратного и программного обеспечения, должны знать язык программирования C. Язык C используется в многих областях компьютерных технологий, таких как разработка программного обеспечения, программирование встроенных систем, моделирование устройств. Умение программировать на C – это важный и востребованный навык.

Следующие разделы содержат полное описание синтаксиса языка C, а также составляющих элементов программы, включая заголовочные файлы, описания функций и переменных, типы данных и часто используемые библиотечные функции. **Раздел 8.6** описывает практическое применение языка C для программирования микроконтроллера PIC32.

Язык C был использован для создания широко распространенных операционных систем – Linux, Windows, OS X, iOS и Android, поскольку язык предоставляет свободный доступ к аппаратным ресурсам компьютера. Сравнивая его с другими языками программирования, допустим Python или Matlab, необходимо отметить, что в языке C отсутствует встроенная реализация таких развитых языковых средств, как высокоуровневые функции для работы с файлами, сравнение по шаблону, матричные операции и взаимодействие с графическим интерфейсом пользователя. Также в нем отсутствуют средства, позволяющие выявить распространенные ошибки при обращении к памяти, подобные выходу за границы массива. Мощь языка C в сочетании с отсутствием контроля за ошибками помогает хакерам проникать в компьютерные системы, где используется программное обеспечение, разработанное без должного внимания к безопасности.

Краткий итог

- **Высокоуровневое программирование**: использование высокоуровневых языковых конструкций удобно при создании широкого круга приложений от программного обеспечения для анализа и моделирования до программ для микроконтроллеров.
- **Низкоуровневый доступ**: преимущество языка C состоит в том, что помимо высокоуровневых конструкций он предоставляет непосредственный доступ к аппаратуре и памяти.

C.2. Добро пожаловать в язык C

Программа на языке C состоит из одного или нескольких текстовых файлов, в которых описываются действия, которые должен выполнить компьютер. Перед выполнением текст программы необходимо перевести в машинное представление, понятное для компьютера. Процесс перевода называется компиляцией. Простая программа, которая выводит фразу «Hello world!» на дисплей, приведена в **примере C.1**. Имена файлов, содержащих текст на языке C, принято заканчивать суффиксом «.c». Хороший стиль программирования подразумевает, что имена файлов будут отражать их содержание. Скажем, файл из **примера C.1** можно назвать `hello.c`.

Пример C.1 ПРОСТАЯ ПРОГРАММА НА C

```
// Вывод "Hello world!" на консоль
#include <stdio.h>

int main(void){
  printf("Hello world!\n");}
```

Вывод

```
Hello world!
```

C.2.1. Структура программы на языке C

Обычно программа на языке C содержит одну или несколько функций. Каждая программа должна иметь функцию с именем main, которая служит стартовой точкой для выполнения программы. Кроме этой главной функции, большинство программ на языке C содержат набор функций, которые находятся в тексте программы и/или в библиотеках. Наш пример `hello.c` состоит из директив включения в программу заголовочных файлов, объявления функции `main` и ее реализации.

Данное приложение дает базовое представление о языке C. Кроме того, существует огромное количество литературы, описывающей язык C более глубоко и детально. Одним из лучших учебников по языку является классическая книга «Язык программирования C» (The C Programming Language), написанная создателем языка Деннисом Ритчи в соавторстве с Брайаном Керниганом, где в концентрированной форме изложены все основные возможности языка C. Еще один хороший учебник — это книга Al Kelley и Ira Pohl «A Book on C».

Заголовочный файл: `#include <stdio.h>`

Заголовочный файл состоит из объявлений функций, требующихся программе. В нашем случае программа использует функцию `printf`, которая находится в стандартной библиотеке ввода/вывода и объявлена в заголовочном файле `stdio.h`. **Раздел C.9** содержит более подробное описание стандартной библиотеки языка C.

Главная функция: `int main(void)`

Работа программы начинается с выполнения операторов, содержащихся в функции с именем `main`. Набор этих операторов называется *телом функции* `main`. Любая программа на С должна иметь одну и только одну функцию `main`. Синтаксис функций описан в **разделе С.6**. Тело функции содержит последовательность операторов, каждый из которых должен заканчиваться точкой с запятой. Ключевое слово `int` указывает, что результат функции (или возвращаемое значение) имеет целочисленный тип. Возвращаемое значение показывает, была ли программа выполнена успешно. По завершении программы результат `main` передается в то окружение, из которого она была запущена.

Тело функции: `printf("Hello world!\n");`

Тело функции `main` из нашего примера содержит единственный вызов функции `printf`, печатающую строку «Hello world!», которая заканчивается символом перевода строки `"\n"`. Более детальное описание функций ввода/вывода содержится в **разделе С.9.1**.

В общих чертах: все программы устроены подобно нашему простому примеру `hello.c`, с той разницей, что сложные программы могут состоять из миллионов строк текста, разбитых на множество функций и файлов.

С.2.2. Запуск С-программы

Программы на языке С могут выполняться процессорами с различными системами команд. Переносимость программ – это еще одно преимущество языка программирования С. Перед запуском программа должна быть скомпилирована С-компилятором. Существует несколько отличающихся реализаций С-компиляторов, включая *cc* (С-компилятор) и *gcc* (GNU С-компилятор). В данной книге мы используем компилятор *gcc* для демонстрации компиляции и запуска программ. Этот компилятор доступен для свободного использования. Он поставляется в составе всех дистрибутивов операционной системы Linux и не требует дополнительных действий по инсталляции. В операционной системе Windows может понадобиться установка пакета программ Cygwin. Компилятор gcc имеется также для большого количества встраиваемых систем, например для микроконтроллеров Microchip PIC32. Процесс подготовки файла с программой, его компиляция и выполнение кода программы описаны ниже. Эти действия одинаковы для всех программ на языке С.

1. Создайте текстовый файл, например `hello.c`.
2. В командной оболочке перейдите в директорию, содержащую файл `hello.c`, затем наберите команду `gcc hello.c` и запустите ее на выполнение.

3. Компилятор создаст файл с исполняемым кодом. По умолчанию этот файл получит имя `a.out` (или `a.exe` в системе Windows).
4. Наберите в командной оболочке `./a.out` (или `a.exe` в системе Windows) и нажмите **Enter**.
5. На экране должна появиться строка «Hello world!».

Краткий итог

- **`filename.c`:** файл с программой на языке C обычно имеет суффикс «.c».
- **`main`:** каждая программа должна содержать только одну функцию `main`.
- **`#include`:** большинство программ на C используют библиотечные функции. Для работы с ними необходимо включить директиву `#include <библиотека.h>` в начало файла с программой.
- **`gcc filename.c`:** файлы с текстом программ преобразуются в исполняемый код при помощи компиляторов, таких как GNU C (gcc) или C (cc) компиляторы.
- **Выполнение программы:** после компиляции программа запускается вводом команды `./a.out` (или `a.exe`) в командной оболочке.

C.3. Компиляция

Компилятор – это программа, которая переводит текст, написанный на языке высокого уровня, в его низкоуровневое представление: язык ассемблера или машинный код. Иначе говоря, компилятор читает файл с программой на языке C и преобразует его в файл с исполняемым кодом. Существует огромное количество литературы по компиляторам, поэтому здесь мы ограничимся только кратким введением. Работу компилятора можно разбить на несколько шагов: (1) препроцессирование, в ходе которого в программу включаются объявления из библиотек и выполняется подстановка макросов; (2) удаление всей не используемой для генерации кода информации, как, например, комментарии; (3) перевод операторов языка высокого уровня в соответствующие им наборы машинных команд; (4) сборка всех файлов с машинными командами, а также библиотечными функциями в единый исполняемый файл. Каждый процессор использует свой набор команд, поэтому программу необходимо скомпилировать именно для того процессора, на котором она будет выполняться. Набор команд для процессора RISC-V описан в **главе 6**.

C.3.1. Комментарии

Программисты используют комментарии для пояснения действий в тексте программы и назначения функций. Всякий, кто видел программы без комментариев, может подтвердить их необходимость. В языке C существуют два вида комментариев: однострочные комментарии начинаются с двух символов `//` и продолжаются до конца строки; многострочные комментарии могут занимать произвольное количество строк. Они начинаются с комбинации символов `/*` и заканчиваются `*/`. Комментарии крайне полезны для структурирования и пояснения текста программы, но во время компиляции они пропускаются и не попадают в исполняемый код.

```
// Это пример однострочного комментария.
/* Это пример
   многострочного комментария. */
```

В начало файла с программой будет полезно вставить комментарий с указанием автора, даты создания, даты последней модификации и назначения программы. Комментарий, приведенный ниже, можно поместить в заголовок файла `hello.c`.

```
// hello.c
// 1 июня 2012 Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// Эта программа печатает "Hello world!" на экране
```

C.3.2. #define

Директива `#define` используется для объявления именованных констант. После объявления именованной константы мы можем использовать в программе ее имя как синоним значения константы. Такие глобально определенные константы также называются *макросами*. Предположим, вы пишете программу, позволяющую пользователю сделать не более 5 попыток дать правильный ответ. В таком случае вы можете использовать директиву `#define` для определения количества попыток в виде макроса.

```
#define MAXGUESSES 5
```

Символ `#` обозначает, что эта строка программы должна быть обработана *препроцессором*. Перед компиляцией препроцессор заменяет все вхождения имени `MAXGUESSES` на значение 5, указанное в дирек-

Числовые константы в языке C по умолчанию считаются десятичными числами. Вы также можете использовать шестнадцатеричные и восьмеричные числа, добавляя перед ними префиксы «0x» и «0». Стандарт языка C99 не определяет запись чисел в двоичной системе исчисления, но они поддерживаются в некоторых компиляторах с использованием префикса «0b». Например, в этих трех строках выполняется присваивание переменной x одного и того же значения:

```
char x = 37;
char x = 0x25;
char x = 045.
```

Неименованную константу, используемую в программе, называют «*магическим числом*». Наличие подобной «магии» приводит к трудноуловимым ошибкам. Например, вы можете изменить константу в одном месте программы, но позабыть внести исправление в другом месте. Глобально определенные именованные константы устраняют «магические числа» из текста программ.

тиве `#define`. Обычно в качестве имен макросов принято использовать прописные буквы и размещать директивы `#define` в начале файла. Объявление именованной константы облегчает дальнейшее сопровождение программы, поскольку можно быть уверенным, что в тексте программы везде используется одно и то же значение. Кроме того, такой подход позволяет быстро изменить все вхождения значения константы в программе. Для этого вам нужно всего лишь исправить одну строчку с директивой `#define` вместо поиска и замены значения константы по всей программе.

В **примере C.2** показано, как использовать директиву `#define` для пересчета дюймов в сантиметры. Переменные `inch` и `cm`, предназначенные для хранения чисел с плавающей запятой, объявлены имеющими тип `float`. Использование именованной константы, объявленной директивой `#define INCH2CM`, может помочь избежать ошибок, вызванных опечатками (например, применение 2,53 вместо 2,54), а также упрощает поиск и замену (например, если нам требуется увеличить точность множителя, используемого для преобразования), особенно если речь идет о большой программе.

Пример C.2 ИСПОЛЬЗОВАНИЕ `#define` ДЛЯ ОБЪЯВЛЕНИЯ КОНСТАНТ

```
// Преобразование дюймов в миллиметры
#include <stdio.h>

#define INCH2CM 2.54
int main(void) {
  float inch = 5.5;     // 5.5 дюйма
  float cm;

  cm = inch *INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Вывод

```
5.500000 inches = 13.970000 cm
```

C.3.3. `#include`

Модульность подразумевает разделение программы на отдельные файлы и функции. Общеупотребительные функции могут быть сгруппированы для упрощения их повторного использования в других программах. Объ-

явления констант, переменных и функций, которые собраны в одном *заголовочном файле*, можно включить внутрь другого файла с помощью директивы препроцессора `#include`. Доступ к функциям *стандартной библиотеки* осуществляется именно таким образом. Например, для использования функций ввода/вывода, таких как `printf`, в программу необходимо включить строку

```
#include <stdio.h>
```

Для имени заголовочного файла обычно используется стандартное расширение `.h`. Директивы `#include` принято размещать в самом начале файла. Хотя в общем случае они могут находиться в любой точке программы, которая предшествует использованию функций, переменных и констант, объявленных в заголовочном файле.

Заголовочный файл, который является частью программы, может быть включен в текст директивой препроцессора `#include`, где имя файла указано в двойных кавычках (`" "`) вместо угловых скобок (`< >`). Например, заголовочный файл `myfunctions.h`, созданный разработчиком программы, включается в текст директивой

```
#include "myfunctions.h"
```

Файлы, указанные в угловых скобках, во время компиляции ищутся в каталогах системных заголовочных файлов. Файлы, указанные в двойных кавычках, ищутся в том же каталоге, где расположен файл, содержащий директиву `#include`. Если заголовочный файл расположен в другом каталоге, необходимо использовать относительный путь до него.

Краткий итог

- **Комментарии:** в языке C используются однострочные (`//`) и многострочные (`/* */`) комментарии.
- **`#define NAME val`:** директива `#define` позволяет использовать идентификатор (NAME) в коде программы. Перед компиляцией идентификатор макроса заменяется на его значение (`val`), указанное в объявлении макроса.
- **`#include`:** директива `#include` позволяет включать в программу объявления библиотечных или общеупотребительных функций, переменных, констант. Для использования объявлений из системных библиотек поместите следующую строку в начало программы: `#include <library.h>`. Для использования заголовочного файла, созданного программистом, используйте двойные кавычки и путь относительно текущего каталога: `#include "other/myFuncs.h"`.

Имена переменных чувствительны к регистру символов и могут выбираться по усмотрению автора программы. Тем не менее существуют некоторые ограничения. Имя переменной не может совпадать с зарезервированными словами, перечисленными в стандарте языка С (`int`, `while` и т. д.). Имя переменной не может начинаться с цифры (например, объявление "`nt 1x;`" некорректно) или включать в себя ряд специальных символов, таких как `\`, `*`, `?` или `-`, хотя символ подчеркивания (_) использовать можно.

Рис. С.1 Как выглядит память для программы на С

С.4. Переменные

Переменные в языке С обладают именем, типом, значением и адресом в памяти. Объявление переменной определяет ее имя и тип. Например, следующее объявление определяет переменную, имеющую тип `char` (размером один байт), и дает этой переменной имя `x`. Адрес, по которому эта однобайтовая переменная будет располагаться в памяти, назначается автоматически при компиляции программы.

```
char x;
```

Память рассматривается в языке С как группа последовательных байтов, в которой каждому байту присвоен уникальный номер, называемый *адресом* (см. **рис. С.1**). Переменная занимает один или несколько байт памяти. Адресом переменной считается адрес первого занимаемого ею байта. Интерпретация набора байтов, отведенных под переменную, зависит от типа переменной. Это может быть целое число, число с плавающей запятой и другие типы данных. Далее мы рассмотрим базовые типы языка С, объявления глобальных и локальных переменных и их инициализацию.

С.4.1. Базовые типы данных

В языке С существует набор базовых (или встроенных) типов данных. В первом приближении их можно разделить на целые числа, числа с плавающей запятой и символы. Для представления целых чисел со знаком используется дополнительный код. Целочисленные типы со знаком и без знака ограничены диапазоном значений. Числа с плавающей запятой также имеют ограниченный диапазон значений и точность. Для их представления используется формат, определенный стандартом IEEE 754. Символы могут рассматриваться как коды ASCII или как 8-битные целые числа. В **табл. С.2** перечислены размеры и диапазоны значений всех базовых типов данных. Целые числа могут занимать 16, 32 или 64 бита в памяти. Для их представления используется дополнительный код, если только переменная не была объявлена с использованием ключевого слова `unsigned`. Размер типа `int` зависит от конкретной платформы и обычно соответствует размеру слова в используемой машине. Например, для 32-битного процессора типы `int` или `unsigned int` имеют размер 32 бита. Для 16-битного процессора тип int обычно имеет размер 16 бит. Но компиляторы для 64-битных процессоров, как правило, используют 32-битные целые числа, чтобы уменьшить ошибки при переносе старого кода, возникающие из-за размера данных. Если важен точный

размер типа данных, используйте `int16_t`, `int32_t` или `int64_t`, чтобы явно определить размер. (Это типы данных со знаком; их аналоги без знака – `uint16_t` и т. д.) Числа с плавающей запятой могут занимать 32 или 64 бита, что соответствует представлению с одинарной или двойной точностью. Символы имеют размер 8 бит.

Таблица C.2 Базовые типы данных и их размеры

Тип	Размер (бит)	Минимум	Максимум
`char`	8	$-2^7 = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32\,768$	$2^{15} - 1 = 32\,767$
`unsigned short`	16	0	$2^{16} - 1 = 65\,535$
`long`	32	$-2^{31} = -2\,147\,483\,648$	$2^{31} - 1 = 2\,147\,483\,647$
`unsigned long`	32	0	$2^{32} - 1 = 4\,294\,967\,295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	аппаратно зависимый		
`unsigned int`	аппаратно зависимый		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Пример C.3 демонстрирует объявления переменных различных типов. Размер памяти, требующийся для хранения переменных, показан на **рис. C.2**. Переменная `x` занимает один байт, переменная `y` занимает два байта, а для переменной `z` требуется четыре байта. Переменные одного типа всегда имеют одинаковый размер, но располагаются в разных участках памяти. В нашем примере переменные `x`, `y` и `z` расположены по адресам 1, 2 и 4 соответственно. Имена переменных чувствительны к регистру символов. Например, имена `x` и `X` обозначают две разные переменные (хотя использование настолько похожих имен переменных в одной программе может привести к путанице).

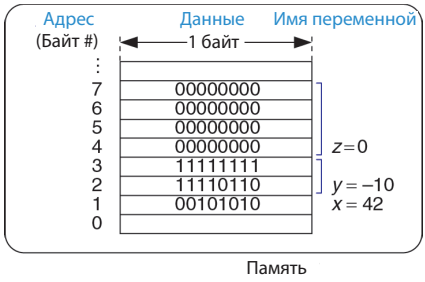


Рис. C.2 Расположение переменных в памяти для примера C.3

Пример C.3 ПРИМЕРЫ ТИПОВ ДАННЫХ

```
// Примеры разных типов данных и их двоичных представлений
unsigned char x = 42;   // x = 00101010
short y = -10;          // y = 11111111 11110110
unsigned long z = 0;    // z = 00000000 00000000 00000000 00000000
```

С.4.2. Глобальные и локальные переменные

Область видимости переменной — это область программы, в пределах которой можно использовать данную переменную. Например, для локальной переменной ее областью видимости является функция, внутри которой объявлена переменная. Локальная переменная будет недоступна вне этой функции.

Глобальные и локальные переменные отличаются местом объявления в программе и областью видимости. Глобальные переменные объявляются вне функций и, как правило, в начале программы. К таким переменным могут обращаться любые функции. Глобальные переменные не рекомендуется использовать слишком часто, так как они нарушают принципы модульности и усложняют понимание программы. Но если переменная необходима сразу нескольким функциям, она может быть сделана глобальной.

Локальная переменная, объявленная в функции, может быть использована только внутри тела этой функции. По этой причине две разные функции могут содержать объявления переменных с одинаковыми именами. Эти переменные не оказывают никакого влияния друг на друга. Они создаются при каждом новом вызове функции и уничтожаются при ее завершении. Данные, хранящиеся в таких переменных, не сохраняются между вызовами функции.

Примеры С.4 и **С.5** позволяют сравнить эти два вида переменных. В первом примере используются глобальные переменные, а во втором — локальные. Глобальная переменная max из **примера С.4** доступна в любой функции. Использование локальных переменных показано в **примере С.5**. Такой подход более предпочтителен, так как он позволяет разделять программу на несколько независимых модулей, взаимодействующих друг с другом через их внешние интерфейсы.

Пример С.4 ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
// использование глобальной переменной для поиска и печати
// максимального из трех чисел

int max; // глобальная переменная, содержащая максимум из трех чисел

void findMax(int a, int b, int c) {
  max = a;
    if (b >max) {
      if (c > b) max = c;
      else       max = b;
    } else if (c > max) max = c;
}

void printMax(void) {
  printf("The maximum number is: %d\n", max);
}

int main(void) {
  findMax(4, 3, 7);
  printMax();
}
```

Пример С.5 ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
// локальные переменные при поиске и печати максимума из трех чисел

int getMax(int a, int b, int c) {
  int result = a; // локальная переменная содержит максимальное значение

  if (b > result) {
    if (c > b) result = c;
    else       result = b;
  } else if (c > result) result = c;
  return result;
}

void printMax(int m) {
  printf("The maximum number is: %d\n", m);
}

int main(void) {
  int max;

  max = getMax(4, 3, 7);
  printMax(max);
}
```

С.4.3. Инициализация переменных

Перед чтением значения переменной она должна быть *проинициализирована*, то есть ей должно быть присвоено некоторое начальное значение. В момент объявления переменной ей выделяется блок памяти, но содержимое этого блока пока не определено. Например, там может находиться значение предыдущей переменной, занимавшей ранее тот же блок памяти, или некое произвольное значение. Глобальные и локальные переменные могут быть проинициализированы либо при их объявлении в программе, либо среди операторов одной из функции. В **примере С.3** продемонстрирована инициализация переменных непосредственно при их объявлении. В **примере С.4** показано, как переменные инициализируются после их объявления, но до первого обращения к ним. Глобальная переменная max инициализируется в функции `getMax` до того, как к ней произойдет первый доступ в функции `printMax`. Чтение значения неинициализированных переменных – это распространенная ошибка, которую бывает довольно трудно обнаружить.

Краткий итог

- **Типы данных:** тип переменной определяет размер (количество байтов, занимаемых переменной) и представление переменной (интерпретацию содержимого памяти). Типы данных, используемые в языке С, перечислены в **табл. С.2**.

- **Память:** в языке C память рассматривается как упорядоченный набор байтов. Значения переменных хранятся в памяти, и каждая переменная имеет свой собственный адрес (номер первого байта).
- **Локальные переменные:** локальные переменные объявляются внутри тела функции и доступны только внутри этой функции.
- **Инициализация переменных:** переменной должно быть присвоено начальное значение до первого обращения к ней. Инициализация переменной может быть выполнена непосредственно при объявлении переменной либо операторами программы.

C.5. Операции

Наиболее распространенный вид конструкций языка C – это *выражения*. Например:

```
y = a + 3;
```

Выражение состоит из *операций* (таких как + или *), выполняющих действия над одним или несколькими *операндами* (переменные или константы). Операции, которые имеются в языке C, перечислены в **табл. C.3**. Они сгруппированы по категориям и перечислены в порядке убывания приоритета операций. К примеру, мультипликативные операции более приоритетны и выполняются до аддитивных. Операции одного уровня приоритета выполняются слева направо в том порядке, как они расположены в выражении.

Таблица C.3 Операции в порядке убывания приоритета

Категория	Операция	Описание	Пример
Унарные	`++`	постинкремент	`a++; // a = a+1`
	`--`	постдекремент	`x--; // x = x-1`
	`&`	получение адреса	`x = &y; // x = адрес y в памяти`
	`~`	побитовое НЕТ	`z = ~a;`
	`!`	логическое НЕТ	`!x`
	`-`	унарный минус	`y = -a;`
	`++`	инкремент	`++a; // a = a+1`
	`--`	декремент	`--x; // x = x-1`
	`(type)`	приведение типа	`x = (int)c; // перевести c в int // и назначить его x`
	`sizeof()`	размер переменной или типа	`long int y; x = sizeof(y); // x = 4`
Мультипликативные	`*`	умножение	`y = x *12;`
	`/`	деление	`z = 9 / 3; // z = 3`
	`%`	модуль (остаток)	`z = 5 % 2; // z = 1`
Аддитивные	`+`	сложение	`y = a + 2;`
	`-`	вычитание	`y = a - 2;`

Таблица C.3 (окончание)

Категория	Операция	Описание	Пример
Побитовый сдвиг	`<<`	побитовый сдвиг влево	`z = 5 << 2; // z = 0b00010100`
	`>>`	побитовый сдвиг вправо	`x = 9 >> 3; // x = 0b00000001`
Отношения	`==`	равенство	`y == 2`
	`!=`	неравенство	`x != 7`
	`<`	меньше	`y <12`
	`>`	больше	`val >max`
	`<=`	меньше или равно	`z <= 2`
	`>=`	больше или равно	`y >= 10`
Побитовые	`&`	побитовое И	`y = a & 15;`
	`^`	побитовое исключающее ИЛИ	`y = 2 ^ 3;`
	`|`	побитовое ИЛИ	`y = a |b;`
Логические	`&&`	логическое И	`x && y`
	`||`	логическое ИЛИ	`x || y`
Тернарные	`? :`	условный оператор	`y = x ? a : b; // если x TRUE, // y=a, иначе y=b`
Присваивание	`=`	присваивание	`x = 22;`
	`+=`	присваиванием с суммированием	`y += 3;  // y = y + 3`
	`-=`	присваиванием с вычитанием	`z -= 10;  // z = z - 10`
	`*=`	присваиванием с умножением	`x *= 4;  // x = x * 4`
	`/=`	присваиванием с делением	`y /= 10;  // y = y / 10`
	`%=`	присваиванием по модулю	`x %= 4;  // x = x % 4`
	`>>=`	присваиванием с побитовым сдвигом вправо	`x >>= 5;  // x = x>>5`
	`<<=`	присваиванием с побитовым сдвигом влево	`x <<= 2;  // x = x<<2`
	`&=`	присваиванием с побитовым И	`y &= 15;  // y = y & 15`
	`|=`	присваиванием с побитовым ИЛИ	`x |= y;  // x = x | y`
	`^=`	присваиванием с побитовым исключающим ИЛИ	`x ^= y;  // x = x ^ y`

Унарные операции имеют только один операнд. Тернарные операции имеют три операнда. А все остальные операции выполняются над двумя операндами и называются бинарными. Тернарная операция (от латинского слова *ternarius*, означающего «состоящий из трех») возвращает второй или третий операнд. Если первый операнд принимает значение TRUE (ненулевое), то результатом выражения будет второй операнд, а иначе если первый операнд – FALSE (нулевое), то результатом станет третий операнд вы-

Правда, только правда и ничего, кроме правды

В языке C считается, что переменная принимает значение TRUE, если она не равна нулю, и FALSE в противном случае. Результат логических и тернарных операций, а также операторов управления последовательностью действий (`if`, `while` и др.) зависит от истинности или ложности значений переменных. Результат операций сравнения или логических операций представляется в виде числа 1 для логического значения ИСТИНА (TRUE) или числа 0 для значения ЛОЖЬ (FALSE).

ражения. **Пример C.6** демонстрирует два способа вычисления максимума из двух значений.

Пример C.6 ТЕРНАРНАЯ ОПЕРАЦИЯ И ЭКВИВАЛЕНТНЫЕ ОПЕРАТОРЫ `IF/ELSE`

```
(a) y = (a > b) ? a : b; // скобки излишни,
                         // но с ними все яснее

(b) if (a > b) y = a;
      else     y = b;
```

В способе (a) используется тернарная операция. Способ (b) немного длиннее, и в нем использованы условные операторы `if/else`.

В составном операторе присваивания сначала выполняется арифметическая операция над операндами, которыми являются левая и правая части оператора, а затем результат вычисления записывается в переменную из левой части составного оператора. **Пример C.7** показывает различные операции языка C. Двоичные значения обозначены префиксом «0b» в комментариях.

Пример C.7 ПРИМЕРЫ ОПЕРАЦИЙ ЯЗЫКА C

Выражение	Результат	Примечание
`44 / 14`	`3`	Результат округлен
`44 % 14`	`2`	Деление по модулю 14
`0x2C && 0xE //0b101100 && 0b1110`	`1`	Логическое И
`0x2C || 0xE //0b101100 || 0b1110`	`1`	Логическое ИЛИ
`0x2C & 0xE //0b101100 & 0b1110`	`0xC (0b001100)`	Побитовое И
`0x2C | 0xE //0b101100 | 0b1110`	`0x2E (0b101110)`	Побитовое ИЛИ
`0x2C ^ 0xE //0b101100 ^ 0b1110`	`0x22 (0b100010)`	Побитовое исключающее ИЛИ
`0xE << 2 //0b1110 << 2`	`0x38 (0b111000)`	Сдвиг влево на 2
`0x2C >> 3 //0b101100 >> 3`	`0x5 (0b101)`	Сдвиг вправо на 3
`x = 14;` `x += 2;`	`x=16`	

`y = 0x2C; // y = 0b101100`
`y &= 0xF; // y &= 0b1111`
	`y=0xC (0b001100)`	

`x = 14; y = 44;` `y = y + x++;`	`x=15, y=58`	Увеличение переменной после использования
`x = 14; y = 44;` `y = y + ++x;`	`x=15, y=59`	Увеличение переменной до использования

C.6. Вызовы функций

Функции также называются процедурами.

Модульность программы является признаком хорошего стиля. Большую программу обычно разделяют на небольшие фрагменты, называемые функциями, которые, по аналогии с аппаратными модулями, имеют хорошо спроектированные входы, выходы и назначение. В **примере C.8** демонстрируется функция `sum3`. Объявление функции начинается с указания типа возвращаемого ею значения (`int`). Затем следует имя функции `sum3`. В конце идет список параметров функции, заключенный в круглые скобки (`int a, int b, int c`). Тело функции заключается в фигурные скобки `{}`. Оно может состоять из последовательности операторов или не иметь вообще ни одного оператора внутри скобок. Оператор `return` определяет результат, который должна вернуть функция. Этот результат можно рассматривать как значение функции на выходе. Функция может вернуть только одно значение.

Пример C.8 ФУНКЦИЯ `SUM3`

```
// возвращает сумму трех входных переменных

int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

В следующем примере, после вызова функции `sum3`, значение переменной `y` будет равно 42.

```
int y = sum3(10, 15, 17);
```

Наличие входных данных и результата функции не является обязательным. **Пример C.9** показывает функцию, у которой нет ни параметров, ни результата на выходе. Ключевое слово `void` перед именем функции означает, что она не возвращает результат. То же ключевое слово `void` между круглыми скобками после имени функции говорит о том, что функция не имеет параметров.

Пример C.9 ФУНКЦИЯ `printPrompt` БЕЗ АРГУМЕНТОВ И РЕЗУЛЬТАТА

```
// печать приглашения на консоли
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

Если между круглыми скобками ничего нет, то это также означает отсутствие у функции входных данных. Таким образом, функция без параметров и результата может быть объявлена следующим образом:

```
void printPrompt()
```

Использование прототипов может не потребоваться при аккуратном выборе порядка расположения функций в тексте программы. Тем не менее при наличии циклических зависимостей между функциями прототипов избежать невозможно. Например, в случае когда функция `f1` вызывает функцию `f2`, которая в свою очередь вызывает функцию `f1`. Размещение прототипов всех функций в начале файла или в отдельном заголовочном файле считается хорошим стилем программирования.

Функция должна быть объявлена текстуально выше того места в программе, где она вызывается. Соблюдения этого правила можно добиться двумя способами. Во-первых, мы можем поместить полное определение функции до того места, где она будет вызвана. Такой способ часто используется, когда функция `main` размещается в конце файла, после объявления всех других функций. Во-вторых, мы можем объявить только *прототип* функции до точки ее вызова и последующего полного определения функции. Прототипом называется первая строчка объявления функции, содержащая тип возвращаемого значения, имя функции и ее параметры. Ниже приведены образцы прототипов функций из **примеров С.8** и **С.9**:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

Пример С.10 демонстрирует использование прототипов функций. Несмотря на то что определения функций размещаются после функции `main`, прототипы функций в начале программы позволяют вызывать функции из главной функции `main`.

Пример С.10 ПРОТОТИПЫ ФУНКЦИЙ

```
#include <stdio.h>

// прототипы функций
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)

{
  int y = sum3(10, 15, 20);
  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c)
{  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Вывод

```
sum3 result: 45
Please enter a number from 1-3:
```

Функция `main` всегда возвращает результат типа `int`. Применительно к главной функции возвращаемое значение служит для сообщения операционной системе о результате работы всей программы. Ноль означает нормальное завершение программы и отсутствие ошибок. Индикатором ошибки служит ненулевое значение результата. Если функция `main` не имеет оператора `return`, она автоматически возвращает ноль в качестве результата. Большинство операционных систем не выводят на экран это возвращаемое значение по завершении программы.

Имена функций точно так же, как имена переменных, чувствительны к регистру символов, и так же не могут быть служебными словами, зарезервированными в языке C. Они не должны начинаться с цифры и включать специальные символы, за исключением символа подчеркивания (_). Обычно в имя функции принято помещать глагол, который говорит о том, что эта функция делает.
Старайтесь придерживаться единого стиля в использовании прописных и строчных букв в именах, чтобы не приходилось постоянно вспоминать, как должно выглядеть имя функции или переменной. Существуют два наиболее распространенных стиля. При использовании первого стиля, называемого camelCase, все слова, составляющие имя функции, пишутся слитно, при этом каждое слово пишется с прописной буквы. Такое имя, например `printPrompt`, напоминает горбы верблюда. Второй стиль характерен использованием символа подчеркивания для разделения слов. К примеру, `print_prompt`. К сожалению, мы обнаружили, что постоянные попытки дотянуться до символа подчеркивания на клавиатуре обостряют приступы туннельного синдрома (мой мизинец начинает болеть, когда я только думаю о подчеркивании). Поэтому мы предпочитаем camelCase. Но самое важное — это придерживаться единого стиля в пределах вашей организации.

C.7. Управление последовательностью выполнения действий

Для управления последовательностью выполнения действий используются конструкции ветвления и циклы. Конструкции ветвления исполняют те или иные наборы операторов в зависимости от некоторого условия. Циклы повторяют набор операторов до тех пор, пока выполняется заданное условие.

C.7.1. Условные операторы

В языке C существует стандартный набор условных операторов `if`, `if/else` и оператор выбора `switch/case`.

Оператор `if`

Оператор `if` выполняет следующий за ним оператор, когда логическое выражение, заданное в скобках, принимает значение ИСТИНА (ненулевое). Синтаксис оператора `if`:

```
if (expression)
  statement
```

В **примере C.11** демонстрируется оператор `if`. Когда переменная `aintBroke` равна 1, переменной `dontFix` также присваивается значение 1. При необходимости выполнения нескольких операторов их группируют в один *составной оператор* (или *блок* кода), для этого используют фигурные скобки `{}`, как показано в **примере C.12**.

Фигурные скобки, `{}`, используются для группирования одного или более операторов в составной оператор или блок.

Пример C.11 ОПЕРАТОР `if`

```
int dontFix = 0;

if (aintBroke == 1)
  dontFix = 1;
```

Пример C.12 ОПЕРАТОР `if` С БЛОКОМ КОДА

```
// Если amt >= $2, спросить пользователя и выдать печеньку

if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

Оператор `if/else`

Конструкция `if/else` выполняет один из двух операторов в зависимости от значения условия, как это показано ниже. Когда выражение `expression` в условии `if` принимает значение ИСТИНА, выполняется оператор `statement1`. В противном случае выполняется оператор `statement2`.

```
if (expression)
  statement1
else
  statement2
```

Использование конструкции `if/else` можно увидеть в **примере C.6 (b)**. Переменной `max` присваивается значение переменной `a`, если `a` больше, чем `b`. В противном случае переменной max присваивается значение `b`.

Оператор `switch/case`

Конструкция `switch/case` выполняет операторы в зависимости от значения в заголовке `switch`. Общая форма записи конструкции `switch`:

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
  default:            statement4
}
```

Например, если значение `variable` совпадает со значением выражения `expression2`, выполнение программы будет продолжено, начиная с оператора `statement2` и до первого встреченного оператора `break`, который завершит выполнение конструкции `switch`. Если не удовлет-

воряется ни одно из условий, то будет выполнен оператор `statement4`, расположенный после ключевого слова `default`.

Если ключевое слово `break` после одного из операторов в теле конструкции `switch` пропущено, то после завершения данного оператора выполнение действий продолжится с переходом на следующую ветвь `case` и т. д. Возможно, это вовсе не то поведение, которое вы ожидаете. Остерегайтесь этой ошибки, распространенной среди начинающих программистов на языке C.

Оператор `switch` используется в **примере C.13**, чтобы присвоить переменной `amt` сумму платежа, которая зависит от значения переменной `option`. Конструкция `switch` эквивалентна серии вложенных операторов `if/else`, как показано в **примере C.14**.

Пример C.13 ОПЕРАТОР `switch/case`

```
// задать amt в зависимости от величины опции
switch (option) {
  case 1: amt = 100;break;
  case 2: amt = 50; break;
  case 3: amt = 20; break;
  case 4: amt = 10; break;
  default: printf("Error: unknown option.\n");
}
```

Пример C.14 ВЛОЖЕННЫЕ ОПЕРАТОРЫ `if/else`

```
// задать amt в зависимости от величины опции
if      (option == 1) amt = 100;
else if (option == 2) amt = 50;
else if (option == 3) amt = 20;
else if (option == 4) amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2. Циклы

Язык C имеет все основные виды операторов цикла, которые встречаются в языках программирования высокого уровня. Это операторы `while`, `do/while` и `for`.

Цикл `while`

Цикл `while` повторяет выполнение оператора, пока заданное условие принимает истинное значение.

```
while (condition)
statement
```

Цикл `while` из **примера C.15** вычисляет значение факториала $9 = 9 \times 8 \times 7 \times ... \times 1$. Обратите внимание, что условие цикла проверяется до выполнения оператора `statement`. В этом примере тело цикла состоит из нескольких операторов, поэтому они заключены в фигурные скобки.

Пример C.15 ЦИКЛ `while`

```
// вычислить 9! (9 факториал)
int i = 1, fact = 1;

// перемножить числа от 1 до 9
while (i <10) { // циклы while сначала проверяют условие
  fact *= i;
  i++;
}
```

Цикл `do/while`

Оператор `do/while` похож на цикл `while`, но только с той разницей, что проверка условия `condition` происходит после выполнения тела цикла `statement`. Общая форма записи оператора `do/while` приведена ниже. Обратите внимание, что после условия `condition` необходима точка с запятой.

```
do
  statement
while (condition);
```

В **примере C.16** цикл `do/while` запрашивает у пользователя ввод числа до тех пор, пока не будет угадано верное значение. Условие станет истинным, когда введенное пользователем число совпадет с заданным значением. Программа проверяет условие только после того, как тело

Пример C.16 ЦИКЛ `do/while`

```
// запрос к пользователю угадать число и проверить его на правильность

#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);  // читать ввод от пользователя
  numGuesses++;
} while ( (numGuesses <MAXGUESSES) & (guess != CORRECTNUM) );
// цикл do проверяет условие после первой итерации

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

цикла do/while будет выполнено. Цикл do/while удобен, когда перед проверкой условия выхода из цикла должны быть выполнены некоторые действия (например, запрос данных у пользователя).

Цикл for

Цикл for подобен циклам while и do/while тем, что он повторяет оператор statement до тех пор, пока выполняется условие цикла. В цикле for принято использовать *переменную цикла*, которая служит для подсчета количества выполненных итераций цикла. Общая форма записи цикла for выглядит так:

```
for (initialization; condition; loop operation)
  statement
```

Инструкция initialization выполняется один раз до начала работы цикла. Условие condition проверяется каждый раз перед началом нового цикла. Если условие не выполняется, то цикл завершается. Выражение loop operation выполняется в конце каждой итерации. **Пример С.17** демонстрирует применение цикла for для вычисления факториала числа 9.

Пример С.17 ЦИКЛ for

```
// Вычисление 9!

int i;  // переменная цикла
int fact = 1;

for (i=1; i<10; i++)
  fact *= i;
```

В **примерах С.15** и **С.16** циклы while и do/while увеличивают значения переменных i и numGuesses на каждой итерации цикла. Аналогичные действия «встроены» в заголовок цикла for. Цикл for может быть переписан с использованием оператора while, но очевидно, что такая форма записи цикла менее удобна.

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

Краткий итог

- **Операторы управления выполнением:** язык C предоставляет операторы управления выполнением, включающие условные операторы и операторы циклов.

- **Условные операторы:** условные операторы выполняют один или несколько операторов, если заданное условие принимает значение TRUE. В языке C имеются следующие условные операторы: `if`, `if/else` и `switch/case`.
- **Циклы:** операторы циклов повторяют выполнение операторов до тех пор, пока условие цикла не примет значение FALSE. Язык C имеет такие средства для организации циклов, как операторы `while`, `do/while` и `for`.

C.8. Другие типы данных

Стандарт языка C определяет целочисленные типы данных различного размера и числа с плавающей запятой. Кроме того, язык включает и другие специальные типы данных, такие как указатели, массивы, строки и структуры. Сейчас мы познакомимся с этими типами данных, а также рассмотрим динамическое выделение памяти.

C.8.1. Указатели

Указатель – это переменная, содержащая адрес (местонахождение в памяти) другой переменной. **Пример C.18** демонстрирует использование указателей. Переменные `salary1` и `salary2` предназначены для хранения целых чисел. Переменная `ptr` может хранить адрес целочисленной переменной. Компилятор назначает каждой переменной некоторый адрес в памяти. Длина адреса зависит от платформы, где программа будет выполняться. Для определенности предположим, что наша программа компилируется на 32-битной системе. Переменная `salary1` занимает участок памяти в диапазоне адресов 0x70–73, переменная `salary2` занимает участок с адресами 0x74–77, а переменная `ptr` находится в диапазоне 0x78–7B. На **рис. C.3** показан блок памяти, выделенный для переменных, а также его содержимое после выполнения программы.

Пример C.18 УКАЗАТЕЛИ

```
// Пример работы с указателем

int salary1, salary2;  // 32-битные числа
int *ptr;              // указатель, выдающий адрес переменной int

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, адрес salary1
salary2 = *ptr + 1000; /* разыменовать ptr на содержимое
                          адреса 70 = $67,500, затем добавить
                          $1,000 и установить salary2 в $68,500 */
```

При объявлении переменной звездочка (*) перед именем переменной обозначает, что переменная является указателем. Если операция * применяется к переменной-указателю в выражении, то она *разыменовывает* указатель, возвращая значение, хранящееся в памяти по адресу, который хранится в переменной-указателе. Когда к переменной применяется операция &, результатом операции будет *адрес* переменной в памяти.

Если переменная-указатель содержит некорректный адрес или адрес, выходящий за диапазон адресов, доступных программе, то, вероятнее всего, разыменовывание такого указателя приведет к аварийному завершению программы. Этот тип ошибки часто называют *ошибкой сегментации*.

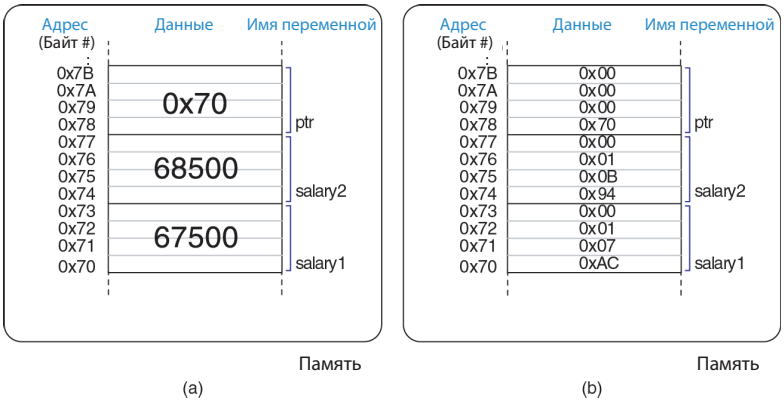


Рис. C.3 Содержимое памяти после выполнения кода из примера C.18: a) значения переменных, b) содержимое ячеек памяти при использовании порядка байтов от младшего к старшему

Указатели особенно полезны, когда функции необходимо изменить значения переданных ей аргументов. В языке C значения аргументов копируются внутрь функции (передача аргументов *по значению*), и поэтому функция не может изменить внешние аргументы напрямую. Вместо этого программист может передать ей указатели на переменные, значения которых нужно модифицировать. Такой способ передачи аргументов называется передачей *по ссылке*. В предыдущих примерах использовалась только передача аргументов *по значению*. В **примере C.19** демонстрируется передача переменной `x` *по ссылке* в функцию `quadruple`, что позволяет функции изменить значение переменной `x`.

Пример C.19 ПЕРЕДАЧА АРГУМЕНТА ФУНКЦИИ ПО ССЫЛКЕ

```
// учетверение величины, на которую указывает указатель
#include <stdio.h>

void quadruple(int *a)

{
```

Пример С.19 (окончание)

```
  *a = *a * 4;
}
int main(void)
{
  int x = 5;
  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Вывод

```
x before: 5
x after: 20
```

Указатель на адрес 0 называется *нулевым указателем* и используется для особого случая, когда переменная не указывает ни на какую другую переменную. Для обозначения такого указателя предназначен системный макрос NULL.

С.8.2. Массивы

В разговорной речи 0-й элемент массива также называется первым элементом массива.

Массивом называется группа однотипных переменных, последовательно расположенных в непрерывном участке памяти. Элементы массива нумеруются, начиная с 0. Если массив состоит из N элементов, то индексом последнего элемента будет $N - 1$. В **примере С.20** объявляется массив `scores`, содержащий результаты экзаменов трех студентов. Для хранения массива выделяется участок памяти, размер которого равен размеру трех переменных типа `long`, что в сумме составляет $3 \times 4 = 12$ байт.

Пример С.20 ОБЪЯВЛЕНИЕ МАССИВА

```
long scores[3];  // массив из трех 4-байтных чисел
```

Предположим, что массив scores располагается в памяти, начиная с адреса 0x40. Адрес первого элемента массива (обозначаемого `scores[0]`) будет 0x40, адрес второго элемента массива – 0x44, а третий элемент массива расположен по адресу 0x48. Распределение памяти показано на **рис. С.4**. Переменная, обозначающая массив, в нашем случае `scores`, эквивалентна указателю на первый элемент массива. Язык С не контролирует доступ к элементам массива, выходящим за его границы. Вся ответственность при работе с элементами массива лежит на разработчике программы.

Программа, модифицирующая данные за границами массива, будет успешно скомпилирована, но ее запуск может привести к непредсказуемым изменениям значений других переменных программы.

Элементы массива можно проинициализировать в момент его объявления, используя фигурные скобки `{}`, как это показано в **примере С.21**. Данный способ инициализации доступен только в точке объявления массива. Альтернативный способ – индивидуальная инициализация элементов массива. Такой способ демонстрируется в **примере С.22**. Для доступа к элементам массива используются квадратные скобки `[]`. Содержимое памяти, выделенной для массива, показано на **рис. С.4**. Цикл `for` является стандартным способом чтения или записи элементов массива, как показано в **примере С.23**.

Пример С.21 ИНИЦИАЛИЗАЦИЯ МАССИВА В МОМЕНТ ОБЪЯВЛЕНИЯ С ИСПОЛЬЗОВАНИЕМ `{}`

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

Пример С.22 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕМ ПРИСВАИВАНИЙ

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

Пример С.23 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕМ ЦИКЛА `for`

```
// Пользователь вводит оценки 3 студентов в массив
long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

Размер массива должен быть известен в момент его объявления. Это необходимо для того, чтобы компилятор мог назначить блок памяти фиксированного размера для хранения массива. В то же время размер массива не играет роли при передаче массива аргументом функции.

Функции необходимо знать только адрес начала массива. **Пример С.24** демонстрирует передачу массива в функцию. Параметр функции `arr` – это просто указатель на первый элемент массива.

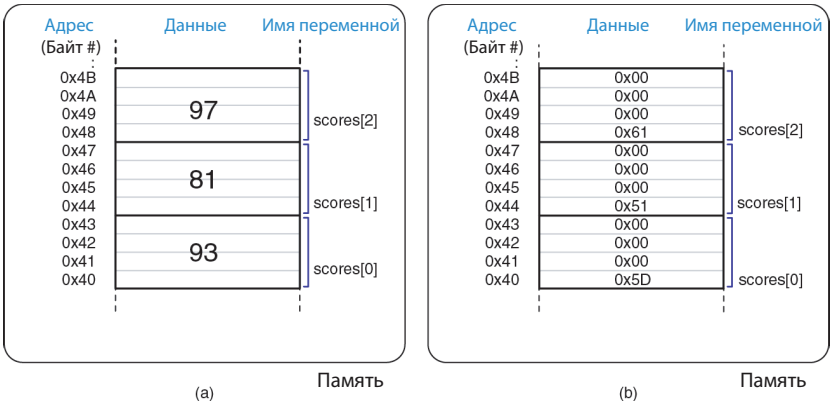


Рис. С.4 Распределение памяти для массива scored

Пример С.24 ПЕРЕДАЧА МАССИВОВ В ФУНКЦИЮ

```
// создать пятиэлементный массив, вычислить среднее и печатать результат
#include <stdio.h>

// возвращает среднее массива (arr) длины len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;
  for (i=0; i <len; i++)
    total += arr[i];
  mean = total / len;
  return mean;
}
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;
  avg = getMean(data, 4);
  printf("The average value is: %f.\n", avg);
}
```

Вывод

```
The average value is: 54.500000.
```

Зачастую количество элементов массива также передается в функцию дополнительным аргументом. Параметр типа `int[]` в объявлении функции указывает на то, что в функцию передается массив элементов типа `int`. Язык С допускает объявления функций, принимающих массивы любого типа.

При объявлении параметров функции указатель на массив эквивалентен указателю на первый элемент массива. Таким образом, функция `getMean` может быть объявлена следующим образом:

```
float getMean(int *arr, int len);
```

Если в функцию передается массив, то предпочтительнее объявлять ее параметр именно как массив. Хотя указатель на первый элемент массива является эквивалентной формой объявления, использование массива подчеркивает ожидаемый тип аргумента.

Функция может возвращать только одно значение. В то же время использование массива в качестве параметра функции позволяет возвращать произвольное количество значений, изменяя внутри тела функции элементы переданного массива. В **примере С.25** функция получает на вход массив целых чисел, сортирует его в порядке возрастания и возвращает результат в том же массиве. Все три объявления функций, приведенных ниже, эквивалентны. Длина массива, указанная в объявлении третьей функции, игнорируется.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

Пример С.25 ПЕРЕДАЧА МАССИВА И ЕГО РАЗМЕРА КАК АРГУМЕНТОВ ФУНКЦИИ

```
// сортировка частей массива vals длиной len по возрастанию
void sort(int vals[], int len)

{
  int i, j, temp;
  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] >vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Массивы в языке С могут иметь более одной размерности. Использование двухмерного массива для хранения результатов решения восьми задач десятью студентами показано в **примере С.26**. Еще раз повторим, что инициализация массива с использованием фигурных скобок `{}` допустима только в момент его объявления.

Пример С.26 ИНИЦИАЛИЗАЦИЯ ДВУХМЕРНОГО МАССИВА

```
// инициализация двумерного массива при объявлении
 int grades[10][8] = { {100, 107, 99, 101, 100, 104, 109, 117},
                       {103, 101, 94, 101, 102, 106, 105, 110},
                       {101, 102, 92, 101, 100, 107, 109, 110},
                       {114, 106, 95, 101, 100, 102, 102, 100},
                       {98,  105, 97, 101, 103, 104, 109, 109},
                       {105, 103, 99,  101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,  107, 98,  101, 109, 104, 110, 108} };
```

Пример С.27 демонстрирует функцию для обработки двухмерного массива с оценками из **примера С.26**. При объявлении функции с многомерным массивом в качестве параметра необходимо указывать размеры всех его измерений, кроме первого. Оба следующих объявления функций являются корректными и эквивалентными:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

Пример С.27 РАБОТА С МНОГОМЕРНЫМИ МАССИВАМИ

```
#include <stdio.h>

// печать содержимого массива 10×8
void print2dArray(int arr[10][8])

{
  int i, j;
  for (i=0; i<10; i++) {         // для каждого из 10 студентов
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // печатать оценки для всех 8 наборов
                                 // задач
    }
    printf("\n");
  }
}

// вычислить среднюю оценку массива 10×8
float getMean(int arr[10][8])
{
  int i, j;
  float mean, total = 0;

  // вычислить среднее в двумерном массиве
  for (i=0; i<10; i++) {
   for (j=0; j<8; j++) {
    total += arr[i][j];   // суммировать величины из массива
   }
```

Пример C.27 (окончание)

```
}
mean = total/(10*8);
printf("Mean is: %f\n", mean);

return mean;
```

Так как массив представляется указателем на его первый элемент, язык C не поддерживает полное поэлементное копирование или сравнение массивов с использованием операций = или ==. Вместо этого вы должны использовать цикл для копирования или последовательного сравнения элементов массива один за другим.

C.8.3. Символы

Для хранения символов в языке C существует тип `char`, представляющий переменные размером 8 бит. Такие переменные могут рассматриваться как числа в дополнительном коде в диапазоне от −128 до 127 или как коды ASCII, обозначающие буквы, цифры или символы. Символы ASCII записываются в виде числовых значений (десятичных, шестнадцатеричных или восьмеричных) или непосредственно как символы, заключенные в одиночные кавычки. К примеру, латинской букве *A* соответствует код 0x41, букве *B* соответствует код 0x42 и т. д. Таким образом, результат выражения `'A' + 3` равен 0x44 или символу `'D'`. **Таблица 6.2** содержит полный набор кодов ASCII. В **табл. C.4** перечислены символы, используемые для форматирования выводимого текста, а также другие специальные символы. Символы форматирования включают в себя возврат каретки (`\r`), перевод строки (`\n`), горизонтальную табуляцию (`\t`), символ конца строки (`\0`). Символ `\r` упомянут для полноты картины, поскольку он используется очень редко. Этот символ возвращает каретку (позицию, куда будет выведен следующий символ) в начало строки (влево). При последующем выводе текста старые символы будут затерты. В отличие от `\r`, символ `\n` передвигает позицию вывода в начало новой

Таблица C.4 **Специальные символы**

Символ	Шестнадцатеричный код	Описание
`\r`	0x0D	Возврат каретки
`\n`	0x0A	Новая строка
`\t`	0x09	Табуляция
`\0`	0x00	Конец строки
`\\`	0x5C	Обратная косая черта (слеш)
`\"`	0x22	Двойная кавычка
`\'`	0x27	Одинарная кавычка
`\a`	0x07	Звонок

Термин «возврат каретки» происходит из эпохи печатных машинок. Для начала печати с левой стороны листа нужно было передвинуть каретку — приспособление, удерживающее лист бумаги, — до упора вправо. Рычаг перевода каретки, который вы можете видеть на фотографии, нажимался таким образом, чтобы каретка сдвигалась вправо и одновременно перемещала лист бумаги по вертикали (т. е. делала перевод строки).

Печатная машинка Ремингтон, использовавшаяся Уинстоном Черчиллем (http://cwr.iwm.org.uk/server/show/conMediaFile.71979)

Строки, использующиеся в языке С, называются строками с завершающим нулем. Их длина определяется поиском нулевого символа в конце строки. Некоторые другие языки программирования, такие как Паскаль, используют иное представление строк. Длина строки в Паскале хранится в ее первом байте. В таком представлении длина строки ограничена 255 байтами. Первый байт называется префиксным байтом, а строки языка Паскаль называют *P*-строками. Преимуществом строк с завершающим нулем является отсутствие ограничений на их размер. Преимущество *P*-строк — в возможности быстрого определения их длины, без необходимости перебора всех символов строки в поисках завершающего нулевого символа.

строки[1]. Нулевой символ *NULL* (`'\0'`) обозначает конец текстовой строки. Мы обсудим его подробнее в **разделе С.8.4**.

С.8.4. Строки символов

Строка — это массив символов, содержащий текст, длина которого ограничена размером массива. При этом длина строки не обязательно должна совпадать с размером массива и может быть меньше. Каждый символ занимает один байт. Для кодирования букв, цифр или символов используется код ASCII. Для обозначения конца строки используется нулевой символ с кодом 0x00. Длину строки можно определить подсчетом символов, начиная с первого и заканчивая предпоследним символом перед нулевым символом конца строки.

В **примере С.28** показано объявление массива `greeting` из десяти символов, содержащего строку `"Hello!"`.

Пример С.28 ОБЪЯВЛЕНИЕ СТРОКИ

```
char greeting[10] = "Hello!";
```

Для определенности предположим, что массив `greeting` расположен в памяти, начиная с адреса 0x50. На **рис. С.5** показано содержимое ячеек памяти с 0x50 до 0x59, включающих строку `"Hello!"`. Обратите внимание, что строка занимает только первые семь элементов массива, хотя в памяти для него выделено десять байт.

Пример С.29 демонстрирует еще один способ объявления строки `greeting`. Указатель `greeting` хранит адрес первого элемента массива, состоящего из семи символов, составляющих строку `"Hello!"`, включая завершающий нулевой символ. Пример также демонстрирует использование функции `printf` и форматирующего спецификатора `%s` для вывода строки на экран.

[1] В операционной системе Windows для обозначения перевода строки используется комбинация символов `\r\n`. В системе UNIX для этого служит одиночный символ `\n`. Будьте внимательны, так как это может быть причиной ошибок при переносе текстов между разными операционными системами.

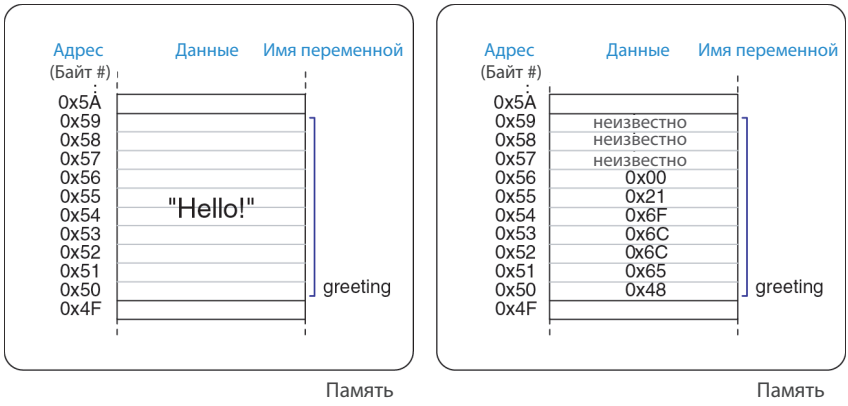


Рис. С.5 **Размещение строки `"Hello!"` в памяти**

Пример С.29 АЛЬТЕРНАТИВНЫЙ СПОСОБ ОБЪЯВЛЕНИЯ СТРОКИ

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Вывод

```
greeting: Hello!
```

В отличие от обычных переменных, строка не может быть скопирована в другую строку прямым присваиванием =. Как и для массивов других типов, каждый символ исходной строки должен быть скопирован индивидуально. В **примере С.30** строка `src` копируется в строку `dst`. При копировании нам не нужно знать размер исходной строки, так как все символы копируются до тех пор, пока не будет найден завершающий нулевой символ. Тем не менее массив `dst` должен быть достаточно большим, чтобы он мог вместить все символы исходной строки. Стандартная библиотека языка С содержат набор функций для работы со строками, и в том числе функцию `strcpy`, используемую для копирования строк (см. **раздел С.9.4**).

Пример С.30 КОПИРОВАНИЕ СТРОК

```
// копия строки-источника src в строку-приемник dst
void strcpy(char *dst, char *src)
{
int i = 0;
  do {
    dst[i] = src[i];       // копировать знаки побайтно
  } while (src[i++]);      // до появления нуль-терминатора
}
```

C.8.5. Структуры

В языке C структуры используются для хранения наборов данных различных типов. Ниже приведена обобщенная форма объявления структуры:

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

Объявление структуры начинается с ключевого слова `struct`. За ним следует имя структуры `name`. Далее в фигурных скобках перечислены компоненты структуры `element1` и `element2`. Структура может содержать произвольное количество компонент. В **примере C.31** объявляется структура `contact`, используемая для хранения контактной информации. Пример демонстрирует объявление переменной `c1`, имеющей тип `struct contact`.

Пример C.31 ОБЪЯВЛЕНИЕ СТРУКТУР

```
struct contact {
  char name[30];
  int phone;
  float height;   // в метрах
};

struct contact c1;
strcpy(c1.name,"Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Вы можете создавать массивы структур и указатели на структуры, используя те же конструкции, что и для встроенных типов данных. В **примере C.32** показано объявление массива структур, хранящих контактные данные.

Пример C.32 МАССИВ СТРУКТУР

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

Общеупотребительной практикой является использование указателей на структуры. Для разыменовывания указателя и доступа к компоненте структуры используется операция `->` (*доступ к компоненте*). В **примере C.33** показано объявление указателя на структуру `contact`, присваивание ему адреса сорок второго элемента массива структур

`classlist` из **примера C.32** и использование операции доступа к компоненте для присваивания нового значения элементу структуры.

Пример C.33 ДОСТУП К КОМПОНЕНТЕ СТРУКТУРЫ С ИСПОЛЬЗОВАНИЕМ УКАЗАТЕЛЯ И ->

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9; // эквивалентно: (*cptr).height = 1.9;
```

Структуры могут быть переданы как аргументы функции или получены из нее по значению или по ссылке. При передаче структуры по значению компилятор копирует все элементы структуры для дальнейшей работы с этой копией внутри функции. Если размер структуры достаточно большой, такой подход может потребовать существенного расхода памяти и времени. Передача структуры по ссылке сводится к передаче в функцию указателя на структуру, и такой способ гораздо эффективнее. При передаче по ссылке функция может также изменить полученную структуру, а не возвращать новую копию. В **примере C.34** демонстрируются два варианта функции `stretch`, уменьшающей значение компоненты структуры `height` на 2 см. Функция `stretchByReference` использует передачу по ссылке, чтобы изменить значение компоненты без двойного копирования всех данных структуры.

Пример C.34 ПЕРЕДАЧА СТРУКТУР ПО ССЫЛКЕ И ПО ЗНАЧЕНИЮ

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}
int main(void)
{
  struct contact George;
  George.height = 1.4; // беднягу вытянули
  George = stretchByValue(George); // растянуть до звезд
  stretchByReference(&George);     // и еще немного
}
```

C.8.6. Оператор `typedef`

Язык C позволяет программисту определять свои собственные имена для типов данных, используя оператор `typedef`. В частности, вместо

использования длинного имени типа `struct contact` мы можем определить новое короткое имя `contact`, как это продемонстрировано в **примере C.35**.

Пример C.35 ОПРЕДЕЛЕНИЕ НОВОГО ИМЕНИ ТИПА ОПЕРАТОРОМ `typedef`

```
typedef struct contact {
  char name[30];
  int phone;
  float height;  // в метрах
} contact;       // определяет contact как подмену "struct contact"

contact c1;      // теперь можно объявить переменную типа contact
```

Оператор `typedef` может быть использован для создания нового типа данных, занимающего тот же объем памяти, что и встроенный тип. В **примере C.36** определяются два новых типа данных `byte` и `bool` размером 8 бит. Использование типа `byte` в объявлении переменной `pos` делает более ясным ее назначение. Она представляет собой 8-битное число, а не символ в коде ASCII. Тип `bool` указывает, что переменная является 8-битным числом с логическими значениями TRUE или FALSE. Пользовательские имена типов делают программу более понятной, в отличие от использования встроенного типа char для любых маленьких переменных.

Пример C.36 ОПРЕДЕЛЕНИЕ ТИПОВ ДАННЫХ `byte` И `bool`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

В **примере C.37** показано использование оператора `typedef` для определения типов `vector` и `matrix`, представляющих собой одномерный массив из трех элементов и двухмерный массив 3×3 соответственно.

Пример C.37 ОПРЕДЕЛЕНИЕ ИМЕН ТИПОВ `vector` И `matrix`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7. Динамическое распределение памяти

Во всех предыдущих примерах память для структур данных выделялась *статически*, так как размер данных был известен в момент компиляции. Такой способ выделения памяти может быть неудобен для массивов и строк переменной длины. Программист вынужден определять размер массива с запасом, чтобы он мог вместить самый большой объем данных, который может встретиться программе. Альтернативный способ – *динамическое* выделение памяти во время выполнения программы. В этом случае мы можем запросить ровно тот объем памяти, который нам нужен.

Функция `malloc`, объявленная в системном заголовочном файле `stdlib.h`, выделяет блок памяти требуемого размера и возвращает указатель на него. При невозможности выделить запрошенное количество памяти функция возвращает NULL. В следующем примере показано выделение памяти для десяти переменных типа `short` (10×2 = 20 байт). Функция `sizeof` возвращает размер в байтах переменной или типа данных.

```
// динамическое выделение 20 байт памяти
short *data = malloc(10*sizeof(short));
```

Пример C.38 демонстрирует динамическое выделение и освобождение памяти. Программа получает произвольное количество чисел, затем сохраняет их в динамически созданном массиве и вычисляет их среднее значение. Количество требуемой памяти зависит от числа элементов массива и размера каждого элемента. Например, если размер типа `int` составляет четыре байта и массив состоит из десяти элементов, необходимо выделить 10 × 4 = 40 байт. Функция `free` освобождает выделенную память, после чего она может быть повторно использована. Динамически выделяемую память необходимо всегда освобождать, а иначе это приведет к *утечкам памяти* в программе.

Пример C.38 ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ПАМЯТИ

```
// динамически выделить и освободить память с помощью malloc и free
#include <stdlib.h>

// Вставьте сюда код функции getMean из примера C.24.

int main(void) {
  int len, i;
  int *nums;

  printf("How many numbers would you like to enter?  ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
```

Пример C.38 ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ПАМЯТИ

```
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:  ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8. Связные списки

Связный список – это популярная структура данных, применяемая для хранения переменного числа элементов. Каждый элемент списка представляет собой структуру, содержащую от одного до нескольких полей с данными и указатель на следующий элемент списка. Первый элемент списка называется *головой* списка. С помощью связных списков можно проиллюстрировать несколько концепций языка C, таких как структуры, указатели и динамическое выделение памяти.

В **примере C.39** связный список используется для хранения списка учетных записей пользователей, размер которого может изменяться. Каждый пользователь имеет имя, пароль, уникальный идентификатор (UID) и флаг, указывающий на обладание административными привилегиями. Связный список содержит структуры типа `userL`, содержащие всю информацию о пользователе и указатель на следующий элемент в списке. Указатель на *голову* списка хранится в глобальной переменной `users`. Его начальное значение NULL указывает на то, что список пустой.

Пример C.39 СВЯЗНЫЙ СПИСОК

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];  // имя пользователя
  char passwd[80]; // пароль
  int uid;         // уникальный идентификатор
  int admin;       // 1 указывает на привилегии администратора
  struct userL *next;
} userL;

userL *users = NULL;
void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;
```

Пример С.39 (окончание)

```
  newUser = malloc(sizeof(userL)); // создать область для нового
                                   // пользователя
  strcpy(newUser->uname, uname);   // скопировать значения в поля
                                   // пользователя
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;    // вставить в начало связанного списка
  users = newUser;
}

void deleteUser(int uid) {  // удалить первого пользователя с данным uid
  userL *cur = users;
  userL *prev = NULL;
  while (cur != NULL) {
    if (cur->uid == uid) {   // поиск удаляемого
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // готово
    }
    prev = cur;      // иначе продолжить сканирование списка
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

В программе объявлены функции для вставки, удаления и поиска пользователя, а также функция для подсчета количества пользователей. Функция `insertUser` выделяет память для нового элемента и добавляет этот элемент в голову списка. Функция `deleteUser` перебирает элементы списка до тех пор, пока не будет найден пользователь с указанным

идентификатором UID. После этого функция удаляет элемент из списка, соединяя его соседей напрямую, и освобождает память, выделенную удаляемому элементу. Функция `findUser` перебирает элементы списка до тех пор, пока не будет найден пользователь с указанным идентификатором UID, и возвращает указатель на найденный элемент или NULL, когда идентификатор не найден. Функция `numUsers` подсчитывает количество элементов в списке.

Краткий итог

- **Указатели:** указатель хранит адрес переменной.
- **Массивы:** массив – это набор однотипных элементов, объявленный с использованием квадратных скобок `[]`.
- **Символы:** тип `char` может хранить небольшие целые числа или коды ASCII, обозначающие текстовые или специальные символы.
- **Строки:** строка – это массив символов. Концом строки является завершающий символ 0x00.
- **Динамическое выделение памяти:** функция `malloc` из стандартной библиотеки служит для выделения блока памяти во время работы программы. Функция `free` освобождает выделенную память.
- **Связные списки:** связный список – распространенная структура данных для хранения переменного числа элементов.

C.9. Стандартная библиотека языка C

Программисты регулярно используют стандартный набор функций, таких как процедуры печати или функции тригонометрических вычислений. Вместо того чтобы изобретать велосипед и писать каждый раз подобные функции заново, язык C предоставляет набор *библиотек*, содержащих часто используемые функции. Каждая библиотека состоит из заголовочного файла и связанного с ним файла скомпилированного объектного кода. Заголовочный файл содержит объявления переменных, функций и определения типов данных. Объектный файл содержит непосредственно код функций. Для получения исполняемого кода объектный код программы должен быть скомпонован с библиотеками, которые используются программой. Подключение библиотек уменьшает общее время компиляции программы, поскольку они не компилируются повторно, а используется уже готовый объектный код. В **табл. C.5** перечислены некоторые наиболее употребительные библиотеки C и дано их краткое описание.

Таблица C.5 Часто используемые файлы стандартной библиотеки

Заголовочный файл библиотеки	Описание
`stdio.h`	**Библиотека ввода/вывода.** Содержит функции печати и чтения данных в/из файла или консоли (`printf`, `fprintf` и `scanf`, `fscanf`) и функции открытия и закрытия файлов (`fopen` и `fclose`)
`stdlib.h`	**Стандартная библиотека.** Содержит функции для генерации случайных чисел (`rand` и `srand`), динамического выделения и освобождения памяти (`malloc` и `free`), завершения программы (`exit`) и преобразований строк в числовые типы данных и обратно (`atoi`, `atoll` и `atof`)
`math.h`	**Математическая библиотека.** Содержит стандартные математические функции, такие как `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor` и `ceil`
`string.h`	**Библиотека функций для работы со строками.** Содержит функции для сравнения, копирования, объединения строк и вычисления длины строки

C.9.1. stdio

Стандартная библиотека ввода/вывода `stdio.h` содержит функции для вывода на консоль, чтения ввода с клавиатуры, чтения и записи файлов. Для использования этих функций заголовочный файл должен быть включен в начало C-файла:

```
#include <stdio.h>
```

Функция printf

Функция `printf` используется для *форматного вывода* текста. Функция принимает один обязательный параметр – строку, заключенную в кавычки " ", и необязательный список переменных, значения которых необходимо вывести на консоль. Строка содержит текст и, опционально, *спецификаторы* форматирования выводимых данных. Перечень имеющихся спецификаторов приведен в **табл. C.6**. **Пример 40** демонстрирует использование функции `printf`.

Таблица C.6 Коды форматирования, используемые функцией `printf`

Код	Формат вывода
`%d`	Десятичное число
`%u`	Беззнаковое десятичное число
`%x`	Шестнадцатеричное число
`%o`	Восьмеричное число
`%f`	Число с плавающей запятой (тип `float` или `double`)
`%e`	Число с плавающей запятой (тип `float` или `double`) в экспоненциальной форме (например, 1,56e7)
`%c`	Символ (тип `char`)
`%s`	Строка (массив символов, завершающийся нулем)

Пример С.40 ИСПОЛЬЗОВАНИЕ `printf` ДЛЯ ВЫВОДА НА КОНСОЛЬ

```
// Простая функция печати
#include <stdio.h>

int num = 42;
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Вывод

```
The answer is 42.
```

По умолчанию типы данных `float` и `double` выводятся с шестью знаками после запятой. Для изменения формата вывода `%f` необходимо заменить на `%w.df`, где *спецификатор ширины* `w` указывает минимальное количество знаков, используемых для вывода числа, а *спецификатор точности* `d` задает минимальное количество символов после десятичной запятой (точки). Обратите внимание, что при подсчете ширины вывода учитывается и десятичная запятая. В **примере С.41** для вывода переменной `pi` используется четыре символа, из них два – для вывода дробной части: 3,14. Для переменной `e` используется восемь символов, три из которых приходятся на дробную часть. Так как целая часть числа содержит только одну цифру, то при выводе она дополняется тремя пробелами до требуемой ширины. Минимальная ширина вывода для переменной `c` составляет пять символов, и три из них – это дробная часть. Но содержимое переменной не укладывается в этот размер, и поэтому минимальная ширина превышается. Выводятся все цифры, составляющие целую часть числа, и три цифры дробной части.

Пример С.41 ФОРМАТИРОВАНИЕ ВЫВОДА ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

```
// Печать чисел с плавающей запятой в различных форматах
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne = %8.3f\nc = %5.3f\n", pi, e, c);
```

Вывод

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Символы `%` и `\` используются как элементы специальных знаков для управления форматом вывода. Поэтому для вывода символов процента и обратного слеша на экран их необходимо удвоить, как показано в **примере С.42**.

Пример C.42 ИСПОЛЬЗОВАНИЕ `printf` ДЛЯ ПЕЧАТИ ЗНАКА ПРОЦЕНТА И ОБРАТНОГО СЛЕША

```
// Вывод символов % и \ на консоль
printf("Here are some special characters: %% \\ \n");
```

Вывод

```
Here are some special characters: % \
```

Функция `scanf`

Функция `scanf` считывает данные, вводимые с клавиатуры. Она использует те же спецификаторы формата, что и функция `printf`. **Пример C.43** демонстрирует использование функции `scanf`. При вызове функции `scanf` выполнение программы приостанавливается в ожидании ввода данных пользователем. Параметрами функции `scanf` является строка с набором спецификаторов формата вводимых данных и указатели на переменные, в которых будут сохранены полученные значения.

Пример C.43 ИСПОЛЬЗОВАНИЕ `scanf` ДЛЯ ЧТЕНИЯ ДАННЫХ С КЛАВИАТУРЫ

```
// Чтение переменных из командных строк
#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;
  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);   // прим.: & не нужен, так как str - указатель
}
```

Чтение и запись файлов

Многим программам требуется читать данные из файлов или записывать в файлы большой объем информации. Перед началом работы с файлом его необходимо открыть, используя функцию `fopen`. Для чтения из файла и записи в файл предназначены две функции `fscanf` и `fprintf`. По завершении всех операций с файлом он должен быть закрыт вызовом функции `fclose`.

Параметрами функции `fopen` являются имя файла и строка символов, определяющая *режим доступа* к файлу. В случае успешного от-

крытия файла функция возвращает указатель на *дескриптор файла*, имеющий тип `FILE*`. Если файл не может быть открыт, то `fopen` возвращает NULL. Это может случиться, когда для чтения данных открывается несуществующий файл или делается попытка открыть для записи файл, в который уже производит запись другая программа. Имеются следующие режимы доступа:

`"w"`: открытие файла для записи. Если файл уже существует, он будет перезаписан;

`"r"`: открытие файла для чтения;

`"a"`: открытие для записи данных в конец файла. Если файла нет, он будет создан.

Вызов функции открытия файла и проверку того, что она вернула ненулевой указатель на дескриптор файла, обычно записывают в одной строке. Такой способ показан в примере С.44. Тем не менее вызов функции `fopen` и проверку результата ее работы можно разбить на две строки:

```
fptr = fopen("result.txt", "w");
if (fptr == NULL)
...
```

Пример С.44 демонстрирует открытие файла, запись в него данных и закрытие файла. Хорошей практикой являются проверка результата открытия файла и выдача пользователю сообщения в случае ошибки. Функция `fprintf` похожа на функцию `printf`. Помимо форматной строки и данных для вывода, она получает указатель на файловый дескриптор. Функция `fclose` сохраняет в файл все данные из файлового буфера, затем закрывает файл и освобождает выделенные системные ресурсы. Функция `exit` будет описана в **разделе С.9.2**.

Пример С.44 ИСПОЛЬЗОВАНИЕ `fprintf` ДЛЯ ЗАПИСИ ДАННЫХ В ФАЙЛ

```
// Запись "Testing file write." в файл result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1); // выйти из программы с указанием ошибки при выполнении
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

Пример С.45 показывает чтение чисел из файла `data.txt` с использованием функции `fscanf`. Перед вызовом этой функции файл должен быть открыт на чтение. Затем программа считывает из файла числа и выводит их на экран. Эти действия повторяются до конца файла. Для определения достижения конца файла служит функция `feof`. По завершении работы программа закрывает файл для освобождения системных ресурсов.

Пример C.45 ИСПОЛЬЗОВАНИЕ `fscanf` ДЛЯ ЧТЕНИЯ ДАННЫХ ИЗ ФАЙЛА

```
#include <stdio.h>

int main(void)
{
  FILE *fptr;
  int data;

  // читать данные из входного файла
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) {  // проверка, что не достигнут конец файла
   fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }
  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Вывод

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Другие полезные функции из библиотеки `stdio`

Функция `sprintf` записывает отформатированные данные в строку символов в памяти. Функция `sscanf` считывает данные из строки. Для чтения одиночного символа из файла используется функция `fgetc`. Функция `fgets` предназначена для чтения из файла целой строки текста.

Возможности функции `fscanf` по чтению данных из файла довольно ограничены. Удобно использовать функцию `fgets` для последовательного чтения файла строка за строкой и применять функцию `sscanf` для обработки прочитанной строки. Еще один способ – чтение и анализ содержимого файла по одному символу, используя вызов `fgetc`.

C.9.2. `stdlib`

Стандартная библиотека `stdlib.h` предоставляет набор функций общего назначения. В ней имеются функции для генерации случайных чисел (`rand` и `srand`), динамического выделения памяти (`malloc` и `free`, обсуждались в **разделе C.8.8** «Динамическое распределение памяти»), функция завершения работы программы (`exit`) и преобразования фор-

матов данных. Чтобы использовать эти функции, необходимо включить следующую директиву в начало программы:

```
#include <stdlib.h>
```

По историческим причинам функция `time` возвращает количество секунд, прошедшее с 00:00 UTC 1 января 1970 года. Аббревиатура UTC обозначает универсальное координированное время, которое может рассматриваться как эквивалент среднего времени по Гринвичу (GMT). Эта дата установлена в честь операционной системы UNIX, разработанной в 1969 году группой сотрудников Bell Labs, включавшей Денниса Ритчи и Брайана Кернигана. По аналогии с празднованием Нового года группы энтузиастов UNIX празднуют наступление моментов времени, соответствующих особым значениям, возвращаемым функцией `time`. Например, 1 февраля 2009 года в 23 часа 31 минуту 30 секунд UTC функция time вернула значение 1 234 567 890. В 2038 году переменные целого 32-битного типа со знаком, использующиеся для хранения времени UNIX, переполнятся и вернутся в 1901 год, так как в типе со знаком интервалы времени могут быть отрицательными.

Функции `rand` и `srand`

Функция `rand` возвращает псевдослучайное целое число. Последовательность псевдослучайных чисел обладает статистическими характеристиками последовательности случайных чисел, но является детерминированной. Конкретный набор чисел зависит от начального значения, называемого ключом, или зерном. Для приведения числа к заданному диапазону используется операция деления по модулю (`%`). **Пример С.46** демонстрирует данный метод для получения случайных чисел в диапазоне от 0 до 9. Значения переменных `x` и `y` – это случайные числа, но они будут одинаковыми при каждом запуске программы.

Пример С.46 ИСПОЛЬЗОВАНИЕ `rand` ДЛЯ ГЕНЕРАЦИИ СЛУЧАЙНЫХ ЧИСЕЛ

```
#include <stdlib.h>
int x, y;

x = rand();         // x = случайное число
y = rand() % 10;    // y = случайное число от 0 до 9
printf("x = %d, y = %d\n", x, y);
```

Вывод

```
x = 1481765933, y = 3
```

Для генерации отличающихся последовательностей случайных чисел программист должен изменять начальное значение ключа. Для этого вызывается функция `srand`, получающая в качестве аргумента новое значение ключа, который должен быть всегда разным. Как показано в **примере С.47**, для получения ключей можно использовать вызов функции `time`, возвращающей текущее время в секундах.

Пример С.47 ИСПОЛЬЗОВАНИЕ `srand` ДЛЯ ИНИЦИАЛИЗАЦИИ ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ

```
// выдает на каждом проходе новое случайное число
#include <stdlib.h>
#include <time.h>   // нужно для вызова time()

int main(void)
```

Пример C.47 (окончание)

```
{
  int x;

  srand(time(NULL)); // запуск генератора случайных чисел
  x = rand() % 10;   // случайное число от 0 до 9
  printf("x = %d\n", x);
}
```

Функция `exit`

Функция `exit` завершает работу программы. Она имеет единственный числовой параметр, который программа возвращает операционной системе. Это число должно сообщать результат выполнения программы. Ноль означает нормальное завершение программы, ненулевое значение обозначает ошибку.

Преобразование форматов: `atoi`, `atol`, `atof`

Стандартная библиотека предоставляет функции для преобразования строк ASCII-символов в числовые типы данных `int`, `long int` и `double`. Это функции `atoi`, `atoll` и `atof`, использование которых показано в **примере C.48**. Помимо прочего, эти функции удобно применять для чтения разнородных типов данных из файла и для разбора аргументов командной строки, представляющих смешанный набор чисел и строк. Работа с командной строкой рассмотрена подробнее в **разделе C.10.3**.

Пример C.48 ПРЕОБРАЗОВАНИЕ ФОРМАТОВ

```
// Преобразование строки ASCII-символов в типы данных int, long и float
#include <stdlib.h>

int main(void)
{
  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");
  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Вывод

```
x = 42 y = 833 z = 3.822000
```

С.9.3. math

Библиотека `math.h` содержит часто используемые функции для математических вычислений. В их число входят тригонометрические функции, процедуры для вычисления корней и логарифмов. Вызовы математических функций демонстрируются в **примере С.49**. Для использования этих функций необходимо включить соответствующий заголовочный файл в начало программы:

```
#include <math.h>
```

Пример С.49 МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

```
// Пример с  математическими функциями
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);          // 1, прим.: входной аргумент в радианах
  b = 2 * acos(0);     // pi (acos значит арккосинус)
  c = sqrt(144);       // 12
  d = exp(2);          // e^2 = 7.389056,
  e = log(7.389056);   // 2 (натуральный логарифм, основание e)
  f = log10(1000);     // 3 (десятичный логарифм, основание 10)
  g = floor(178.567);  // 178, округлить до ближайшего меньшего целого числа
  h = pow(2, 10);      // вычисление 2 в десятой степени

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f,
          g = %.2f, h = %.2f\n", a, b, c, d, e, f, g, h);
}
```

Вывод

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00,
h = 1024.00
```

С.9.4. string

Библиотека `string.h` предоставляет широкий набор функций для манипуляций со строками. Ниже приведены основные функции из этой библиотеки:

```
// копирует строку src в строку dst и возвращает dst
char *strcpy(char *dst, char *src);

// объединяет (добавляет) src в конец строки dst и возвращает dst
char *strcat(char *dst, char *src);

// сравнивает две строки. Возвращает 0, если строки совпадают,
// и ненулевое значение в противном случае
int strcmp(char *s1, char *s2);

// возвращает длину строки, не включая завершающий нулевой символ
int strlen(char *str);
```

C.10. Компилятор и опции командной строки

До этого момента мы рассматривали достаточно простые С-программы. В реальном мире программы могут состоять из десятков или даже тысяч файлов. Это требуется для удобства работы с исходными текстами, модульной организации программы и обеспечения совместной работы множества программистов. В этом разделе мы обсудим использование компилятора для генерации кода программы из множества исходных файлов, а также рассмотрим опции компилятора и аргументы командной строки.

C.10.1. Компиляция нескольких исходных С-файлов

Чтобы скомпилировать сразу несколько файлов с исходными текстами, вы должны перечислить имена всех файлов в командной строке при запуске компилятора. Помните, что в программе должна быть только одна функция `main`, которая обычно находится в файле `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2. Опции компилятора

Опции компилятора позволяют программисту задать такие параметры, как имя и формат генерируемого файла, настройки оптимизации и т. д. Имена и назначение опций не стандартизированы, но в **табл. C.7** перечислены опции, используемые наиболее часто. Каждая опция начинает-

Таблица C.7 Опции компилятора

Опция	Описание	Пример
`-o` файл	Задает имя выходного файла	`gcc -o hello hello.c`
`-S`	Компиляция программы в файл на языке ассемблера (неисполняемый)	`gcc -S hello.c` на выходе получаем `hello.s`
`-v`	Расширенная диагностика — вывод расширенных данных о ходе компиляции	`gcc -v hello.c`
`-Olevel`	Уровень оптимизации программы (обычно уровень принимает значения от 0 до 3). Повышение уровня позволяет получить более быстрый и/или маленький код в обмен на скорость компиляции	`gcc -O3 hello.c`
`--version`	Вывод версии компилятора	`gcc -version`
`--help`	Вывод всех опций компилятора	`gcc --help`
`-Wall`	Вывод всех предупреждений при компиляции	`gcc -Wall hello.c`

ся с символа тире (-). Например, опция -o позволяет указать свое имя файла с исполняемым кодом вместо имени a.out, которое будет дано компилятором по умолчанию. Количество опций очень велико. Полный список опций можно получить, запустив в консоли команду

```
gcc -help.
```

C.10.3. Аргументы командной строки

Как и другие функции, main может принимать входные аргументы. Аргументы для main перечисляются в командной строке при запуске программы. **Пример C.50** демонстрирует обработку командной строки. Параметр argc называется *счетчиком аргументов* и содержит число аргументов, переданных программе. Параметр argv называется *вектором аргументов* и является массивом строк, указанных в командной строке. Предположим, что программа из **примера C.50** компилируется в файл testargs. Если запустить ее, как показано в примере ниже, значение параметра argc будет равно четырем, а массив argv будет содержать следующие значения: {"./testargs", "arg1", "25", "lastarg!"}. Обратите внимание, что первым идет имя самой программы. Результат выполнения программы показан после **примера C.50**.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Если программе требуются числовые аргументы, то она может получить их из строковых значений с использованием функций для преобразования форматов данных, объявленных в stdlib.h.

Пример C.50 АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ

```
// Печатать аргументы командной строки
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i])
;}
```

Вывод

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

С.11. Типичные ошибки

Как и с любым другим языком программирования, вы наверняка будете допускать ошибки при написании нетривиальных программ. Далее мы перечислим некоторые из наиболее распространенных ошибок, допускаемых при программировании на языке С. Часть из них способна доставить особенно серьезные неприятности, так как программа с этими ошибками будет успешно скомпилирована, но результат ее работы окажется неверным.

Ошибка С.1 ПРОПУЩЕННЫЙ & В ВЫЗОВЕ `scanf`

Ошибочный код

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // перед a
                // пропущен &
```

Правильный код

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

Ошибка С.2 ИСПОЛЬЗОВАНИЕ = ВМЕСТО == ДЛЯ СРАВНЕНИЯ

Ошибочный код

```
if (x = 1)  // всегда принимает
            // значение TRUE
  printf("Found!\n");
```

Правильный код

```
if (x == 1)
  printf("Found!\n");
```

Ошибка С.3 ИНДЕКС ПОСЛЕДНЕГО ЭЛЕМЕНТА МАССИВА

Ошибочный код

```
int array[10];
array[10] = 42; // индекс должен быть
в диапазоне 0-9
```

Правильный код

```
int array[10];
array[9] = 42;
```

Навыки отладки приходят с практикой. Тем не менее мы можем дать несколько советов.

- Начинайте исправлять код с первого сообщения об ошибке, выданного компилятором. Последующие сообщения могут быть вызваны наведенным эффектом от первой ошибки. После проверки текста запустите компиляцию снова и продолжайте работу над текстом программы до тех пор, пока все ошибки, обнаруживаемые компилятором, не будут исправлены.
- Если вы не видите ошибки в строке, на которую указывает компилятор, посмотрите на предыдущие строки. Возможно, там пропущена точка с запятой после оператора.
- Разделяйте длинные и сложные выражения на несколько строк.
- Используйте функцию `printf` для печати промежуточных результатов программы.
- Если результат программы не соответствует ожидаемому, начинайте отладку кода с самого раннего места, где обнаружилось отклонение в данных.
- Обращайте внимание на все предупреждения компилятора. Некоторые из них можно проигнорировать, но другие могут указывать на скрытые проблемы, которые приводят к ошибочной работе программы.

Ошибка С.4 ИСПОЛЬЗОВАНИЕ = В ДИРЕКТИВЕ `#define`

Ошибочный код

```
// это приведет к замене NUM на строку "= 4"
#define NUM = 4
```

Правильный код

```
#define NUM 4
```

Ошибка С.5 ИСПОЛЬЗОВАНИЕ НЕИНИЦИАЛИЗИРОВАННОЙ ПЕРЕМЕННОЙ

Ошибочный код

```
int i;
if (i == 10) // переменная i не инициализирована
  ...
```

Правильный код

```
int i = 10;
if (i == 10)
  ...
```

Ошибка С.6 НЕПОЛНЫЙ ПУТЬ К ПОЛЬЗОВАТЕЛЬСКОМУ ЗАГОЛОВОЧНОМУ ФАЙЛУ

Неполный путь к файлу

```
#include  "myfile.h"
```

Правильный путь к файлу

```
#include  "othercode\myfile.h"
```

Ошибка С.7 ИСПОЛЬЗОВАНИЕ ЛОГИЧЕСКИХ ОПЕРАЦИЙ (`!`, `||`, `&&`) ВМЕСТО БИТОВЫХ ОПЕРАЦИЙ (`~`, `|`, `&`)

Ошибочный код

```
char x=!5;    // логическое НЕТ: x = 0
char y=5||2;  // логическое ИЛИ: y = 1
char z=5&&2;  // логическое И: z = 1
```

Правильный код

```
char x=~5;   // поразрядное НЕТ: x = 0b11111010
char y=5|2;  // поразрядное ИЛИ:  y = 0b00000111
char z=5&2;  // поразрядное И: z = 0b00000000
```

Ошибка С.8 ЗАБЫТЫЙ `break` В ОПЕРАТОРЕ `switch/case`

Ошибочный код

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default:  direction = 0;}
// direction = 0
```

Правильный код

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default:  direction = 0;
}
// direction = 2
```

Ошибка С.9 ПРОПУЩЕННЫЕ ФИГУРНЫЕ СКОБКИ `{}`

Ошибочный код

```
if (ptr == NULL) // пропущены фигурные скобки
   printf("Unable to open file.\n");
   exit(1);      // выполняется независимо от условия
```

Правильный код

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

Ошибка С.10 ИСПОЛЬЗОВАНИЕ ФУНКЦИИ ДО ЕЕ ОБЪЯВЛЕНИЯ

Ошибочный код

```
int main(void)
{
  test();}

void test(void)
{...
}
```

Правильный код

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

Ошибка С.11 ОБЪЯВЛЕНИЕ ГЛОБАЛЬНЫХ И ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ С ОДИНАКОВЫМ ИМЕНЕМ

Ошибочный код

```
int x = 5;  // объявление глобальной переменной x
int test(void)
{
  int x = 3; // объявление локальной переменной x
  ...
}
```

Правильный код

```
int x = 5;  // объявление глобальной переменной x
int test(void)
{
  int y = 3; // объявление локальной переменной y
  ...
}
```

Ошибка С.12 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕМ { } ПОСЛЕ ЕГО ОБЪЯВЛЕНИЯ

Ошибочный код

```
int scores[3];
scores = {93, 81, 97}; // ошибка компиляции
```

Правильный код

```
int scores[3] = {93, 81, 97};
```

Ошибка С.13 ПРИСВАИВАНИЕ МАССИВОВ С ИСПОЛЬЗОВАНИЕМ =

Ошибочный код

```
int scores[3]= {88, 79, 93};
int scores2[3];

scores2 = scores;
```

Правильный код

```
int scores[3]= {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
  scores2[i] = scores[i];
```

Ошибка С.14 ПРОПУЩЕННАЯ ТОЧКА С ЗАПЯТОЙ ПОСЛЕ ЦИКЛА do/while

Ошибочный код

```
int num;
do {
  num = getNum();
} while (num <100)  // пропущена ;
```

Правильный код

```
int num;
do {
  num = getNum();
} while (num <100);
```

Ошибка С.15 ИСПОЛЬЗОВАНИЕ ЗАПЯТОЙ ВМЕСТО ТОЧКИ С ЗАПЯТОЙ В ЦИКЛЕ `for`

Ошибочный код

```
for (i=0, i <200, i++)
 ...
```

Правильный код

```
for (i=0; i <200; i++)
 ...
```

Ошибка С.16 ДЕЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ ВМЕСТО ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Ошибочный код

```
// целочисленное деление (с отбрасыванием остатка)
// происходит, когда оба операнда операции деления
// являются целыми числами
float x = 9 / 4;    // x = 2.0
```

Правильный код

```
// для получения числа с плавающей запятой
// хотя бы один из операндов операции деления
// должен иметь тип float или double
float x = 9.0 / 4;  // x = 2.25
```

Ошибка С.17 ЗАПИСЬ ПО НЕИНИЦИАЛИЗИРОВАННОМУ УКАЗАТЕЛЮ

Ошибочный код

```
int *y = 77;
```

Правильный код

```
int x, *y = &x;
*y = 77;
```

Ошибка С.18 ЗАВЫШЕННЫЕ ОЖИДАНИЯ ИЛИ НАДЕЖДА НА УДАЧУ

Начинающий программист часто пишет программу в виде длинной последовательности операторов и при этом надеется, что его программа сразу заработает и будет выполняться правильно. Но реальная практика показывает, что любую нетривиальную программу лучше разбить на короткие фрагменты, оформленные в виде функций, и протестировать каждую из них. С ростом размера программы сложность и время ее отладки возрастают экспоненциально.

Другая распространенная ошибка – надежда на удачу. Это происходит, когда программист, закончив подготовку текста, запускает программу и проверяет, что она работает, но при этом не проверяет корректность результатов, выдаваемых программой. Очень важно выполнить тестирование программы, когда на вход программе подаются разные заранее подготовленные наборы входных данных и затем фактические результаты сравниваются с ожидаемыми результатами.

Это приложение ориентировано на использование языка С для написания программ, выполняемых в системах, подобных компьютерам с операционной системой Linux. **Раздел 8.6** описывает применение языка С для программирования микроконтроллеров PIC32, используемых во встраиваемых системах. Язык С используется для программирования микроконтроллеров благодаря тому, что он обеспечивает практически такой же низкоуровневый доступ к оборудованию, как и язык ассемблера. Но в отличие от языка ассемблера программы на языке С более лаконичны и пишутся быстрее.

Дополнительная литература

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

Захватывающая биография Роберта Нойса, одного из изобретателей интегральной микросхемы и основателя Fairchild Semiconductor и Intel. Для каждого, кто думает о работе в Кремниевой долине, книга даст понимание культуры региона, культуры, на которое Нойс оказал большее влияние, чем любой другой человек.

Ciletti M., *Advanced Digital Design with the Verilog HDL*, 2nd ed., Prentice Hall, 2010.

Хорошее справочное пособие по Verilog 2005 (но не по SystemVerilog).

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.

История разработки нескольких поколений процессоров Pentium, рассказанная одним из руководителей проекта. Для тех, кто рассматривает карьеру в этой области, книга покажет, как выглядит управление огромными проектами и вид из-за кулис на разработку одной из наиболее значимых линеек микропроцессоров.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

Наиболее полный текст по машинной арифметике. Великолепный ресурс по построению высококачественных арифметических устройств для компьютеров.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach, 6th ed.*, Morgan Kaufmann, 2017.

Авторитетная книга по компьютерной архитектуре высокой производительности. Если вы интересуетесь работой передовых микропроцессоров, эта книга для вас.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

Классическая история о создании компьютерной системы. Три десятилетия спустя история по-прежнему захватывает, а изложенные секреты управления проектами все так же актуальны.

Patterson D., and Waterman A., *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon, 2017.

Краткое введение в архитектуру RISC-V от двух архитекторов RISC-V.

Pedroni V., *Circuit Design and Simulation with VHDL, 2nd ed.*, MIT Press, 2010.

Справочное пособие, где хорошо показано, как создавать схемы на языке VHDL.

SystemVerilog IEEE Standard (IEEE STD 1800).

IEEE-стандарт по SysemVerilog HDL; последнее обновление в 2012.
Доступен по ссылке: ieeexplore.ieee.org.

VHDL IEEE Standard (IEEE STD 1076).

IEEE-стандарт по VHDL; последнее обновление в 2008.
Доступен по ссылке: ieeexplore.ieee.org.

Wakerly J., *Digital Design: Principles and Practices, 5th ed.*, Pearson, 2018.

Всеобъемлющее и хорошо написанное руководство по цифровой схемотехнике и отличный справочник.

Weste N., and Harris D., *CMOS VLSI Design, 4th ed.*, Addison-Wesley,

Разработка СБИС (VLSI) – это наука и искусство по построению микросхем, содержащих множество транзисторов. Эта книга, написанная в соавторстве с одним из наших любимых авторов, охватывает область разработки СБИС от основ и до самых передовых приемов, используемых в коммерческих продуктах.

Предметный указатель

Л

М

Н

О

П

Р

С

Т

У

Ф

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Дэвид М. Харрис, Сара Л. Харрис

Научный редактор русского перевода А. Ю. Романов

Цифровая схемотехника и архитектура компьютера: RISC-V

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В. С., Романов А. Ю.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А., Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура QuantAntiqua. Печать цифровая.
Усл. печ. л. 65,81. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Предыдущие издания бестселлера Харрисов уже помогли исправить серьезный дисбаланс в преподавании цифровой электроники в России. Книга также стала отправной точкой для создания курса практических работ на ПЛИС под эгидой МИЭМ НИУ ВШЭ, онлайн-курсов от РОСНАНО и семинаров на ChipEXPO в Сколково. Новое издание Харрисов выходит как раз тогда, когда в России разворачиваются амбициозные проекты по созданию высокопроизводительных процессорных ядер, совместимых с открытой архитектурой RISC-V и при этом разработанных в России.
Мы ожидаем, что читатели данной книги станут топ-разработчиками и бизнес-лидерами российской электронной промышленности и помогут ей занять место в мире, которое соответствует российским традициям».

Юрий Панчул,
инициатор проекта перевода трех книг Харрисов,
инженер-проектировщик CPU, GPU, с опытом работы в MIPS Technologies, Imagination Technologies, Juniper Networks и Samsung Advanced Computing Lab

Книга представляет собой введение в современное проектирование цифровых микросхем. Она предназначена прежде всего будущим архитекторам, разработчикам и верификаторам чипов, но может быть полезна всем, кто хочет понять, что происходит на уровнях технологии между физикой и программированием.

Основные темы:

- схемотехника: комбинационные и последовательностные схемы и их временные параметры;
- языки описания аппаратуры SystemVerilog и VHDL;
- система команд RISC-V;
- микроархитектура, аппаратная организация процессоров. Примеры однотактной, многотактной и конвейерной реализации RISC-V. Читатель сможет разработать свой процессор и реализовать его на микросхемах ПЛИС;
- программирование RISC-V чипа от компании SiFive и интеграция его с периферией, чтобы читатель мог сравнить учебный процессор из предыдущих глав с промышленным.

На сайте издательства www.dmkpress.com можно скачать коды всех примеров программ на SystemVerilog и VHDL, а также другие полезные материалы к книге.

ELSEVIER

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

Интернет-магазин :www.dmkpress.com

Оптовая продажа:
КТК «Галактика». books@alians-kniga.ru

ISBN 978-5-97060-961-3