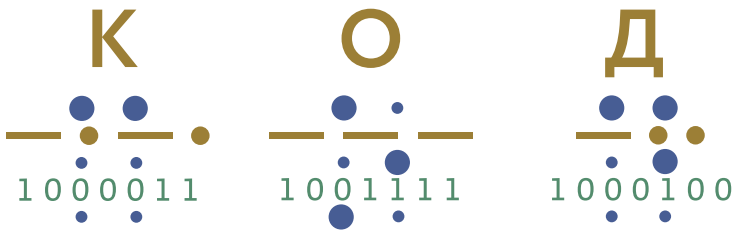
Тайный язык информатики



Чарльз Петцольд

Эту книгу хорошо дополняют:

Неизбежно
12 технологических трендов,
которые определяют наше будущее
Кевин Келли

Верховный алгоритм
Как машинное обучение изменит наш мир
Педро Домингос

Кому нужна математика?
Понятная книга о том, как устроен цифровой мир
Нелли Литвак и Андрей Райгородский

Невероятные приключения Лавлейс и Бэббиджа
(Почти) правдивая история первого компьютера
Сидни Падуа

Charles Petzold

Code

The Hidden Language of Computer Hardware and Software

MICROSOFT PRESS
2000

Чарльз Петцольд

Код

Тайный язык информатики

Перевод с английского Олега Сивченко

Москва
«Манн, Иванов и Фербер»
2019

УДК 681.3
ББК 32.973
П31

Научные редакторы Валерий Артюхин, Азат Гизатулин

Издано с разрешения Pearson Education, Inc.

Книга рекомендована к изданию Дмитрием Воротилиным, Юрием Коровкиным, Александром Самохваловым, Ольгой Сосминой

Петцольд, Чарльз

П31 Код. Тайный язык информатики / Чарльз Петцольд ; пер. с англ. О. Сивченко ; [науч. ред. В. Артюхин, А. Гизатулин]. — М. : Манн, Иванов и Фербер, 2019. — 448 с.

ISBN 978-5-00117-545-2

Книга «Код» представляет собой увлекательное путешествие в прошлое — мир электрических устройств и телеграфных машин. Знакомство с прообразами первых компьютеров позволит читателю с любым уровнем технической подготовки узнать о том, как работают современные электронные устройства.

УДК 681.3
ББК 32.973

ISBN 978-5-00117-545-2

Оглавление

Предисловие к оригинальному изданию . . . 9
Глава 1. Лучшие друзья . . . 12
Глава 2. Коды и комбинации . . . 18
Глава 3. Брайль и двоичные коды . . . 23
Глава 4. Устройство фонарика . . . 30
Глава 5. Заглядывая за угол . . . 41
Глава 6. Телеграфы и реле . . . 50
Глава 7. Наши десять цифр . . . 57
Глава 8. Альтернативы десятке . . . 64
Глава 9. За битом бит . . . 80
Глава 10. Логика и переключатели . . . 98
Глава 11. Логические вентили . . . 117
Глава 12. Двоичный сумматор . . . 147
Глава 13. А как насчет вычитания? . . . 160
Глава 14. Обратная связь и триггеры . . . 174
Глава 15. Байты и шестнадцатеричные числа . . . 202
Глава 16. Сборка памяти . . . 213
Глава 17. Автоматизация . . . 231
Глава 18. От счетов к микросхемам . . . 268
Глава 19. Два классических микропроцессора . . . 294
Глава 20. Набор символов ASCII . . . 326
Глава 21. Шины . . . 345
Глава 22. Операционная система . . . 368
Глава 23. Фиксированная точка, плавающая точка . . . 387
Глава 24. Языки высокого и низкого уровня . . . 403
Глава 25. Графическая революция . . . 421
Благодарности . . . 444
Об авторе . . . 445

Предисловие к оригинальному изданию

Замысел «Кода» я вынашивал лет десять. И тогда, и во время работы над рукописью, и даже когда книга вышла из типографии многие спрашивали: «О чем она?»

Я всегда отвечал уклончиво, бормотал что-нибудь в духе: «Необычная экскурсия по истории цифровых технологий, сформировавших современную эпоху» — в надежде, что этого будет достаточно. Но в какой-то момент мне пришлось признать: «Код» — это книга о том, как устроены компьютеры.

Как я и опасался, отклики были неблагожелательными. На возражение в духе: «А-а, у меня уже есть такая книга» — я немедленно парировал: «Отнюдь, такой — нет». И по-прежнему так считаю. «Код» не похож на прочие книги «о компьютерах». В нем нет больших цветных иллюстраций с дисководами, где стрелками показано, как данные поступают в компьютер, нет рисунков, где паровозик в товарных вагончиках везет нули и единички. Метафоры и сравнения чудесны в своей буквальности, но они ни на что не годны, лишь затмевают красоту технологий.

Мне говорили: «А кому интересно, как работают компьютеры?» Верное замечание. Мне, например, нравится вникать в устройство приборов, но я хочу сам решать, когда это делать. Так, описать, как работает мой холодильник, я смогу лишь под пыткой.

Однако окружающие часто задают вопросы, свидетельствующие об их интересе к внутреннему устройству компьютера. Типичный пример: «Чем отличается оперативная память от дисковой?» Естественно, это важный вопрос. Такие понятия составляют основу маркетинга ПК. Даже начинающему пользователю требуется знать, сколько *мегов* одного и *гигов* другого потребуется для конкретного приложения. Кроме того, новичок должен представлять, что такое файл, как он загружается с диска в память, а затем сохраняется там.

На вопрос о дисковой и оперативной памяти принято отвечать: «Память похожа на столешницу, а диск — на ящики стола». В принципе неплохой ответ, но мне он кажется неудовлетворительным. Создается впечатление, будто архитектура компьютера разрабатывалась по образу и подобию бюро. На самом деле

разница между оперативной и дисковой памятью — искусственная и обусловлена отсутствием единого энергонезависимого и при этом быстро работающего носителя. Так называемая архитектура фон Неймана, доминирующая в компьютерной индустрии уже более 50 лет, возникла в результате этого технического изъяна. Когда меня спрашивали, как запускать программы для Macintosh под Windows, я впадал в ступор, осознавая, что для ответа придется затронуть массу технических тонкостей, которые собеседник явно сразу не поймет.

Хочу, чтобы с помощью «Кода» вы научились разбираться во всех этих вещах настолько, чтобы смогли потягаться с электротехниками и программистами. Надеюсь, вы оцените, каким достижением является компьютер среди технологий XX века, и прочувствуете его красоту саму по себе, без метафор и сравнений.

По сути, компьютеры иерархичны: на самом нижнем уровне располагаются транзисторы, а венчает все информация, которая выводится на монитор. В книге мы будем придерживаться этой иерархии. В принципе, книга и структурирована от уровня к уровню. И этот путь не столь сложен, как может показаться. Да, в современном компьютере происходит масса всякой всячины, но это самые обычные и простые операции.

Хотя в настоящее время компьютеры сложнее, чем четверть или полвека назад, они не изменились фундаментально. Вот почему изучать историю техники так здорово: чем сильнее углубляешься в прошлое, тем проще становятся технологии. Именно поэтому легко добраться до точки, где понятно решительно все.

В книге «Код» я заглянул настолько далеко в прошлое, насколько смог. Сам поразился, что удалось добраться до XIX века и на примере первых телеграфных машин объяснить устройство компьютера. Теоретически все, о чем говорится в первых 17 главах, легко собирается из простейших электрических компонентов, которые в ходу уже более века.

Думаю, благодаря всей этой винтажной технике при чтении вы испытаете некоторую ностальгию. Книгу «Код» невозможно было бы озаглавить «Еще быстрее, еще технологичнее» или «Сверхскоростной бизнес на цифровых нейронах»: определение бита дается лишь на 79-й странице, байта — на 199-й. Транзисторы впервые упоминаются на 156-й странице, и то вскользь.

Итак, пусть «Код» и весьма основательно объясняет устройство компьютера (найдется немного других книг, где описано, например, как именно работает процессор), стиль книги вполне развлекательный. Несмотря на глубину темы, я старался устроить читателю максимально комфортную прогулку. Без всяких вагончиков с нулями и единицами.

Чарльз Петцольд
16 августа 2000 года

[Код]

3a. Система сигналов для представления букв и цифр при передаче сообщений.

b. Система символов, букв или слов, которым присваиваются некоторые произвольно подобранные значения. Используется для передачи сообщений в случаях, когда требуется добиться конфиденциальности или краткости.

4. Система символов, применяемая для представления компьютерных команд…

Словарь английского языка американского наследия

Глава 1
Лучшие друзья

Вам десять лет. Ваш лучший друг живет на другой стороне улицы, напротив. Даже окна ваших спален обращены друг к другу. Каждый вечер родители объявляют отбой в безбожно ранний час, а вам еще хочется пообщаться, поделиться мыслями, наблюдениями, секретами, сплетнями, шутками и мечтами. Никто не вправе вас за это упрекнуть. В конце концов, стремление к коммуникации — одно из наиболее характерных человеческих качеств.

Пока в спальнях горит свет, можно помахать друг другу из окон и, полагаясь на примитивный язык тела, жестикулируя, обменяться парой мыслей. Однако передавать таким образом сложную информацию вряд ли удастся. И как только родители скомандуют: «Погаси свет!» — ситуация кажется безнадежной.

Как общаться? Может, по телефону? А был ли у вас в комнате телефон, когда вам было десять? Даже если так, то, где бы он ни находился, вас подслушают. Если ваш домашний компьютер подключен к телефонной линии, возможно, через него удастся поболтать бесшумно, но, опять же, компьютера в комнате нет.

Однако у вас с другом есть карманные фонарики. Все знают, что такой фонарик изобрели специально для чтения книжки под одеялом, но он отлично подходит для ночной коммуникации. Такая связь практически бесшумна, а луч света бьет прицельно, и, пожалуй, его не заметишь в щель под дверью. Бдительные домашние ничего не заподозрят.

Можно ли общаться при помощи вспышек? Попробовать точно стоит. В первом классе вы учились писать на бумаге слова и буквы, поэтому кажется уместным экстраполировать эти знания на обмен сигналами. Просто встаньте у окна и попытайтесь рисовать буквы светом. Чтобы написать *O*, включите фонарик, опишите им круг в воздухе, а потом выключайте. *I* — это вертикальная палочка. Но, как вы вскоре убедитесь, этот метод просто не работает. Наблюдая за фонариком друга, которым тот выводит в воздухе буквы, вы поймете, как сложно мысленно скомпоновать эти штрихи во что-то цельное. Завитушки и мазки света не слишком *точны*.

Наверняка вы видели в фильмах, как два морехода сигнализировали друг другу над водой мерцающими фонариками, один шпион покачивал зеркальцем, направляя свет сообщнику в другую комнату. Кажется, вот решение.

Сначала разработаем простой метод. Каждая буква алфавита соответствует последовательности бликов. Таким образом, один блик будет означать *А*, два — *Б*, три — *В* и т. д. Для *Я* уже понадобятся 33 блика. Слово «ГДЕ» — 4 блика + 5 бликов + 6 бликов, которые передаются с небольшими паузами, чтобы не перепутать эту серию с 15 бликами, то есть с *Н*. Паузы-пробелы между словами должны быть чуть длиннее.

Но вот что скажу: махать фонариком в воздухе больше не понадобится. Достаточно направить его куда нужно и нажимать на кнопочку. Но здесь возникает другая проблема: одно из первых сообщений, которое вы решите отправить («Как дела?»), растянется на 44 вспышки! Более того, придется забыть о пунктуации, ведь неизвестно, сколько бликов соответствуют вопросительному знаку.

Однако вы уже у цели. Вы предполагаете, что кто-то уже сталкивался с такой проблемой. Рано утром вы отправляетесь в библиотеку на поиски и узнаёте о чудесном изобретении под названием «азбука Морзе». *Вот* то, что нужно, пусть даже теперь придется переучиваться, как пишутся все буквы алфавита.

В чем разница: в изобретенной вами системе каждой букве алфавита соответствует определенное количество бликов, от 1 для *А* до 33 для *Я*. В азбуке Морзе два вида бликов: краткие и долгие. Разумеется, при этом код Морзе получается сложнее, но на практике оказывается гораздо эффективнее. Теперь словосочетание «Как дела?» состоит всего из 24 бликов, а не из 44, причем *с учетом* кода вопросительного знака.

Обсуждая принцип работы азбуки Морзе, принято говорить не о долгих и кратких бликах, а о точках и тире, поскольку при помощи этих знаков удобно изображать код на печатной странице. В азбуке Морзе каждой букве алфавита соответствует краткая серия точек и тире, показанная в таблице на следующей странице.

Пусть азбука Морзе и не связана с компьютером, она помогает познать суть кода, а это важная предпосылка для глубокого понимания тайных языков и внутреннего устройства компьютерного харда и софта.

В этой книге слово «*код*» обычно означает систему передачи информации между людьми и машинами. Иными словами, код обеспечивает коммуникацию. Иногда покажется, что код — это шифр, но большинство кодов таковыми не являются, хотя и должны быть понятными, поскольку лежат в основе человеческого общения.

В начале романа «Сто лет одиночества» Габриэль Гарсия Маркес вспоминает времена, когда «мир был еще таким новым, что многие вещи не имели

названия, и на них приходилось показывать пальцем». Создается впечатление, что лексемы присваиваются понятиям совершенно произвольно. Сложно понять, почему собаку называют собакой, а кошку — кошкой. Можно сказать, что словарь — это своеобразный код.

A	• —	R	• — •	А	• —	Р	• — •
B	— • • •	S	• • •	Б	— • • •	С	• • •
C	— • — •	T	—	В	• — —	Т	—
D	— • •	U	• • —	Г	— — •	У	• • —
E	•	V	• • • —	Д	— • •	Ф	• • — •
F	• • — •	W	• — —	Е	•	Х	• • • •
G	— — •	X	— • • —	Ё	•	Ц	— • — •
H	• • • •	Y	— • — —	Ж	• • • —	Ч	— — — •
I	• •	Z	— — • •	З	— — • •	Ш	— — — —
J	• — — —			И	• •	Щ	— — • —
K	— • —			Й	• — — —	Ъ	— — • — —
L	• — • •			К	— • —	Ы	— • — —
M	— —			Л	• — • •	Ь	— • • —
N	— •			М	— —	Э	• • — • •
O	— — —			Н	— •	Ю	• • — —
P	• — — •			О	— — —	Я	• — • —
Q	— — • —			П	• — — •		

Звуки, которые мы произносим и складываем в слова, — код, понятный любому, кто слышит наш голос и понимает язык, на котором мы говорим. Этот код называется говорением, или речью. Существуют и другие коды для записи слов на бумаге (камне, дереве, в воздухе, например когда самолет выводит рекламные надписи в небе). Такой код — это и рукописные и печатные символы, которые мы видим в книгах, журналах или газетах. Мы называем его письменной речью, текстом. Во многих языках речь и текст согласуются друг с другом. Например, в английском буквы и буквосочетания (в большей или меньшей степени) соответствуют произносимым звукам.

Для глухих или немых был разработан иной код, облегчающий межличностное общение, — язык жестов, состоящий из движений рук, передающих отдельные буквы, слова или целые концепции. Для слепых письменный текст заменяется азбукой Брайля — системой выпуклых точек, соответствующих буквам, буквосочетаниям или целым словам. Когда приходится быстро записывать речь, удобно пользоваться стенографией или сокращениями.

При общении мы пользуемся различными кодами, поскольку одна кодировка удобнее других. Например, устную речь невозможно хранить на бумаге,

и ее заменяет письмо. Тихо передавать информацию на расстоянии невозможно ни при помощи речи, ни на бумаге. Удобная альтернатива — азбука Морзе. Далее мы увидим, что в компьютерах применяются различные типы кодов для передачи чисел, звуков, музыки, изображений и видео. Компьютер не может работать непосредственно с человеческими кодами: машина не в состоянии сымитировать работу человеческих глаз, ушей, рта и пальцев. Недавно* в компьютерной технике наметилась такая тенденция: настольные ПК собирают и хранят различные виды информации, используемой при человеческом общении, и имеют возможность манипулировать такой информацией и ее отображениями. Это визуальная (текст, картинки) и акустическая (речь, звуки, музыка) информация, их комбинация (анимация или кино). Для всех этих типов требуются собственные коды, точно так же как при разговоре используются одни органы (рот и уши), а при письме и чтении — другие (руки и глаза).

Даже сама таблица с азбукой Морзе (с. 14) — в некотором роде код. В таблице каждая буква представлена последовательностью точек и тире. Но как передать точки и тире? Получается, они соответствуют бликам. Для обозначения точки мы быстро перещелкиваем кнопку фонарика (короткий блик), тире — задерживаем фонарик включенным чуть дольше. Так, чтобы передать *A*, мы быстро перещелкиваем фонарик, а потом включаем и выключаем его более медленно. Перед отправкой следующего символа делаем небольшую паузу. Принято, что тире должно быть примерно втрое длиннее точки. Так, если точка длится одну секунду, то тире — три (на самом деле азбука Морзе транслируется гораздо быстрее). Адресат видит короткий сигнал, затем длинный и понимает, что это *A*.

Паузы между точками и тире в азбуке Морзе критически важны. Так, при передаче *A* фонарик должен быть выключен между точкой и тире в течение периода, по длительности примерно равного одной точке. (Если точка длится одну секунду, то промежуток между точкой и тире также длится секунду.) Между буквами в слове выдерживаются более долгие паузы, сравнимые по длительности с тире (в данном случае по три секунды). Например, вот так на азбуке Морзе будет «привет» (обратите внимание на паузы между буквами).

• — — • • — • • • • — — • —

Между словами выдерживается период длительностью примерно два тире (шесть секунд, если тире — три секунды). Вот код фразы «как дела».

— • — • — — • — — • • • • — • • • —

* Книга написана в 2000 году, и автор отталкивается от реалий того периода. *Прим. перев.*

Длительность периодов, в течение которых фонарик остается включен или выключен, не фиксируется. Все периоды отсчитываются относительно длительности точки, а эта длина зависит от того, как быстро удается перещелкивать фонарик, насколько быстро отправитель азбуки Морзе успевает вспомнить код для той или иной буквы. Тире у быстрого отправителя может получиться таким же коротким, как точка у неторопливого. Из-за этой небольшой проблемы расшифровка сообщений может усложняться, но после первых двух-трех букв адресат обычно успевает сориентироваться, где точка, а где тире.

На первый взгляд, определение кода Морзе — под *определением* в данном случае я понимаю соответствие различных последовательностей точек и тире буквам алфавита — кажется столь же произвольным, как и раскладка клавиатуры на пишущей машинке. Если присмотреться, не все так однозначно. Сравнительно простые и краткие коды присваиваются более частотным буквам алфавита, например *E* и *T**. Любители игр «Эрудит» и «Поле чудес» могли это сразу приметить. У редких букв (например, *Q* и *Z* на латинице, за которые в «Эрудите» присваивается по 10 очков) коды длиннее.

Практически каждый хоть немного знает азбуку Морзе. Три точки, три тире, три точки — SOS, международный сигнал бедствия. SOS не аббревиатура. Это просто код из азбуки Морзе, который легко запоминается. Во время Второй мировой войны Британская радиовещательная компания предваряла некоторые передачи первыми нотами из Пятой симфонии Бетховена: ТА-ТА-ТА-ТАММММ! Сочиняя эту музыку, Людвиг ван Бетховен еще не мог знать, что именно такая последовательность сигналов (точка-точка-точка-тире) в азбуке Морзе будет соответствовать букве *V*, с которой начинается английское слово victory — «победа».

Один из недостатков азбуки Морзе в том, что в ней нет капитализации букв. Однако она позволяет передавать не только буквы, но и цифры, которым соответствуют свои последовательности по пять точек и тире.

1	• — — — —	6	— • • • •
2	• • — — —	7	— — • • •
3	• • • — —	8	— — — • •
4	• • • • —	9	— — — — •
5	• • • • •	0	— — — — —

Эти коды как минимум чуть более регулярны, чем буквенные. Для большинства знаков препинания используются по пять, шесть или семь точек и тире.

* В русском языке (по массивам текстов) это буквы *О, Е, А*. *Прим. науч. ред.*

.	• • • • • •	'	• — — — — •
,	• — • — • —	(	— • — — • —
?	• • — — • •	)	— • — — • —
:	— — — • • •	=	— • • • —
;	— • — • —	+	• — • — •
-	— • • • • —	$	• • • — • • —
/	— • • — •	¶	• — • — • •
"	• — • • — •	–	• • — — • —

Кроме того, существуют дополнительные коды для букв с диакритическими знаками из некоторых европейских языков и специальные последовательности-сокращения. Одно из таких сокращений — код SOS. Его следует посылать непрерывно, делая между каждой тройкой символов паузу в одну точку.

Вы убедитесь, что общаться с другом азбукой Морзе гораздо удобнее, если вооружиться специальным фонариком. Кроме обычного переключателя-ползунка, на такой фонарик монтируется кнопочный переключатель, который мы нажимаем и отпускаем, и фонарик зажигается и гаснет. Напрактиковавшись, вы, вероятно, научитесь передавать и принимать по пять-десять слов в минуту, что все равно гораздо медленнее, чем речь (при разговоре в минуту укладывается около 100 слов*), но вполне неплохо.

Когда вы с другом наконец-то выучите азбуку Морзе (а иначе общение при помощи этих сигналов не построить), вы сможете пользоваться таким словарем и в обычной речи. Для максимально быстрого общения произносите точку как «*дих*» («*дит*», если это последняя буква в слове), а тире — как «*дах*». Подобно тому как азбука Морзе позволяет сократить письмо до точек и тире, устный код редуцирует речь всего до двух слогов.

В данном случае ключевой элемент — *двойка*. Два типа бликов, два слога. Два любых феномена, если они разные, в правильных комбинациях подходят для передачи информации.

---

* При речи на английском языке. В русском языке темп речи (скорость произнесения ее элементов) медленнее, поскольку слова на 20–30% длиннее. *Прим. науч. ред.*

Глава 2
Коды и комбинации

Азбуку Морзе придумал Сэмюэл Финли Бриз Морзе (1791–1872). Это изобретение неотделимо от создания телеграфа, о работе которого нам также предстоит узнать. Азбука Морзе послужила хорошим вводным материалом для знакомства с сущностью кода, а телеграф — такой же удобный пример, иллюстрирующий аппаратное обеспечение компьютера.

Многим кажется, что азбуку Морзе проще передавать, чем принимать. Даже если вы не знаете ее на память, можете просто сверяться с таблицей, где буквы для удобства расставлены по алфавиту (с. 14).

Принимать азбуку Морзе и переводить ее в обычные слова значительно сложнее и дольше, поскольку вы работаете в обратном порядке: выясняете, какая буква соответствует конкретной кодовой последовательности точек и тире. Например, если вы получите сигнал «тире-точка-тире-тире», придется заглянуть в таблицу и просмотреть почти все буквы одну за другой, пока не выяснится, что перед вами *Ы*.

Проблема в том, что у нас есть таблица для следующего перевода:

буква алфавита → последовательность азбуки Морзе, состоящая из точек и тире.

Однако нет обратной таблицы:

последовательность азбуки Морзе, состоящая из точек и тире, → буква алфавита.

В начале изучения азбуки Морзе такая таблица, безусловно, пригодилась бы. Правда, не вполне понятно, как ее составить. Точки и тире не допускают никакого подобия алфавитного порядка.

Давайте забудем об алфавите. Пожалуй, разумнее сгруппировать коды таким образом, чтобы их расстановка зависела от количества точек и тире в той

или иной букве. Так, последовательность из азбуки Морзе, содержащая одну точку и одно тире, может означать всего одну из двух букв: *E* или *T*.

E	•	E	•
T	—	T	—

Комбинации, в которых содержится по два знака (либо точки, либо тире), дают нам уже четыре буквы: *И, A, H* и *M*.

I	• •	И	• •
A	• —	A	• —
N	— •	H	— •
M	— —	M	— —

Паттерн из трех символов, точек или тире, дает нам восемь букв: *C, Д, У, K, P, Г, O, B*.

S	• • •	D	— • •	C	• • •	Д	— • •
U	• • —	K	— • —	У	• • —	K	— • —
R	• — •	G	— — •	P	• — •	Г	— — •
W	• — —	O	— — —	B	• — —	O	— — —

Наконец (если мы хотим прекратить это упражнение, пока не перешли к цифрам и знакам препинания), четырехзначные последовательности точек и тире дают нам еще 16 символов.

H	• • • •	B	— • • •	X	• • • •	Б	— • • •
V	• • • —	X	— • • —	Ж	• • • —	Ь	— • • —
F	• • — •	C	— • — •	Ф	• • — •	Ц	— • — •
Ü	• • — —	Y	— • — —	Ю	• • — —	Ы	— • — —
L	• — • •	Z	— — • •	Л	• — • •	З	— — • •
Ä	• — • —	Q	— — • —	Я	• — • —	Щ	— — • —
P	• — — •	O	— — — •	П	• — — •	Ч	— — — •
J	• — — —	Ş	— — — —	Й	• — — —	Ш	— — — —

Всего в этих таблицах содержится 2 + 4 + 8 + 16 кодов суммарно для 30 букв; это на четыре кода больше, чем требуется для полной латиницы, состоящей из 26 букв. Именно поэтому четыре кода в последней таблице отведены под буквы с диакритическими знаками.

Эти четыре таблицы помогут с легкостью переводить любые сообщения, передаваемые азбукой Морзе. Получив код конкретной буквы, вы считаете, сколько в нем точек и тире, и решаете, с какой из таблиц сверяться. Каждая таблица устроена так, что код, состоящий из одних точек, располагается в верхнем левом углу, а код из одних тире — в нижнем правом углу.

Замечаете закономерность в *размерах* четырех таблиц? Обратите внимание: в каждой следующей таблице вдвое больше кодов, чем в предыдущей. Это логично: в последующей таблице содержатся все коды из предыдущей «плюс точка», а также все коды из предыдущей «плюс тире».

Эту тенденцию можно резюмировать следующим образом.

Количество точек и тире	1	2	3	4
Количество кодов	2	4	8	16

Каждая из четырех таблиц содержит вдвое больше кодов, чем предшествующая ей таблица, так что если в первой таблице 2 кода, то во второй — 2 × 2 кодов, в третьей — 2 × 2 × 2 кодов. Вот как еще можно это представить.

Количество точек и тире	1	2	3	4
Количество кодов	2	2 × 2	2 × 2 × 2	2 × 2 × 2 × 2

Разумеется, при умножении числа самого на себя можно использовать степени. Так, 2 × 2 × 2 × 2 можно записать как 2^4 (2 в четвертой степени). Числа 2, 4, 8 и 16 являются степенями двойки, поскольку представляют произведения, которые можно получить умножением двойки самой на себя. Итак, нашу таблицу можно переписать и так.

Количество точек и тире	1	2	3	4
Количество кодов	2^1	2^2	2^3	2^4

Таблица сильно упростилась. Количество кодов равно просто 2 в степени <количество точек и тире>. Можно резюмировать табличные данные в виде простой формулы:

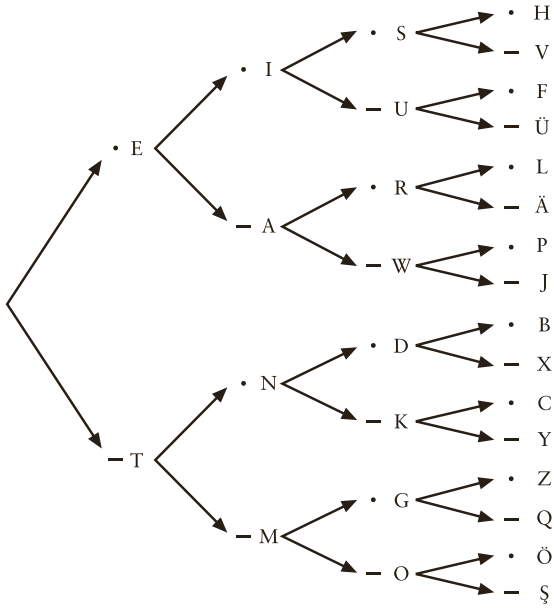
$$\text{Количество кодов} = 2^{\text{количество точек и тире}}.$$

Степени двойки часто используются в различных кодах (другой пример рассмотрим в следующей главе).

Чтобы еще сильнее упростить расшифровку кода Морзе, давайте попробуем построить большую древовидную схему на следующей странице.

На схеме показано, какие буквы получаются при постепенном усложнении последовательностей точек и тире. Чтобы расшифровать конкретную последовательность, идите по стрелкам слева направо. Допустим, мы хотим выяснить, какая буква соответствует коду «точка-тире-точка». Начинаем слева, берем точку; далее идем по стрелкам, выбираем тире, а затем еще одну точку. Получаем букву *R*, расположенную около последней точки.

Такая схема необходима прежде всего для того, чтобы определить код Морзе. Во-первых, она страхует от тупой ошибки: не дает присвоить двум разным буквам один и тот же код. Во-вторых, вы гарантированно задействуете все возможные коды, не выстраивая чрезмерно длинных последовательностей из точек и тире.



Рискуя получить схему, которая не поместится на печатной странице, мы могли бы расширить ее и добавить туда пятизначные коды из точек и тире. Последовательность из пяти точек и тире даст нам 32 ($2 \times 2 \times 2 \times 2 \times 2$, или 2^5) дополнительных кода. Как правило, этого достаточно не только для букв, но и для 10 цифр и 18 знаков препинания, включаемых в азбуку Морзе: цифры действительно кодируются пятизначными последовательностями точек и тире. Правда, многие другие пятизначные коды зарезервированы не за знаками препинания, а за буквами с диакритическими знаками.

Чтобы система учитывала все знаки препинания, в нее нужно включить последовательности из шести точек и тире. Таким образом получим

64 ($2 \times 2 \times 2 \times 2 \times 2 \times 2$, или 2^6) дополнительных кода для суммарного множества из 2 + 4 + 8 + 16 + 32 + 64, или 126, символов. Для азбуки Морзе этого слишком много, поэтому большинство таких длинных кодов остаются неопределенными. Слово «*неопределенный*» в данном контексте указывает на код, который ничего не означает. Если бы вы, принимая азбуку Морзе, получили неопределенный код, то могли бы почти не сомневаться, что кто-то просто допустил ошибку.

У нас хватило смекалки построить эту небольшую формулу:

$$\text{Количество кодов} = 2^{\text{количество точек и тире}}.$$

Так давайте продолжим нашу таблицу и посмотрим, сколько кодов получится из более длинных последовательностей точек и тире.

Количество точек и тире	**Количество кодов**
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

К счастью, нет необходимости выписывать все возможные коды, чтобы определить, сколько их будет. Достаточно умножать двойку на себя нужное количество раз.

Код Морзе называется *двоичным* (что буквально означает «два на два»), поскольку любой его элемент включает только два компонента: точку и тире. Такой код подобен монете, которая может упасть только решкой или орлом. Двоичные объекты (например, монеты) и двоичные коды (например, азбука Морзе) всегда можно описать в виде степеней двойки.

Проделанный нами анализ двоичных кодов — это простое упражнение в одной математической дисциплине, которая называется *комбинаторикой,* или *комбинаторным анализом*. Традиционно комбинаторный анализ особенно активно используется в теории вероятностей и статистике, поскольку связан с выявлением количества вариантов комбинаций различных объектов (например, монет или игральных костей). Он также помогает понять, как составляются и разбираются коды.

Глава 3
Брайль и двоичные коды

Сэмюэл Морзе не был первым, кому успешно удалось транслировать буквы письменного языка в интерпретируемый код. Он не был первым и среди тех, чья фамилия запомнилась как название кода, а не имя собственное. Такая честь выпала слепому французскому подростку, родившемуся примерно через 18 лет после Морзе, но оставившему след в истории гораздо раньше. О жизни Луи Брайля известно немногое, но это захватывающая история.

Луи Брайль родился в 1809 году во французском городке Кувр**е**, в 40 километрах к востоку от Парижа. Отец мальчика был шорником. Будучи трех лет от роду (а в таком возрасте дети не должны играть в отцовской мастерской), Луи случайно ткнул себе в глаз шорным ножом. В ране начался процесс заражения, инфекция распространилась и на второй глаз, и мальчик полностью ослеп. Наверняка его ждала жизнь в невежестве и бедности (как и большинство слепцов в те времена), но Луи проявил незаурядный ум и тягу к знаниям. Благодаря участию деревенского пастора и школьного учителя Луи ходил в сельскую школу вместе с другими ребятами, а в возрасте десяти лет отправился в Парижский государственный институт для слепых детей.

Разумеется, одна из главных сложностей при обучении незрячих в том, что они не могут читать печатные книги. Основатель этой парижской школы Валентин Гаюи (1745–1822) изобрел систему тисненых выпуклых букв для чтения их на ощупь. Но пользоваться системой было сложно, и вышло только несколько книг, напечатанных таким методом. Гаюи не смог посмотреть глубже. Для него буква *A* оставалась буквой *A*. Она должна была выглядеть (ощущаться) как *A*. (Общаясь на языке световых сигналов, мы пробовали рисовать буквы в воздухе и убедились, что такой прием неработоспособен.) Вероятно, Гаюи не догадался, что некий код, сильно отличающийся от печатного алфавита, оказался бы для незрячих удобнее.

Прообраз такого альтернативного кода возник в достаточно необычном контексте. Шарль Барбье, капитан французской армии, изобрел систему записи под названием écriture nocturne, или «ночная азбука». В ней использовались

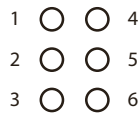
узоры выпуклых точек и тире на плотной бумаге. Предполагалось, что солдаты могли бы обмениваться в темноте такими записками, когда требовалось соблюдать тишину. Писать точки и тире можно было специальным стилусом, вроде шила. Затем выпуклые точки можно было читать на ощупь. Недостаток системы Барбье заключался в ее чрезмерной сложности. Комбинации точек и тире соответствовали звукам, а не буквам алфавита, поэтому одно слово часто могло шифроваться разными кодами. Система хорошо работала для обмена короткими сообщениями в полевых условиях, но решительно не подходила для сравнительно крупных текстов, тем более книг.

Луи Брайль познакомился с системой Барбье в двенадцатилетнем возрасте. Ему понравились выпуклые точки не только потому, что они легко читались на ощупь, но и потому, что их было просто *писать*. Ученик в классе, вооружившись бумагой и стилусом, в самом деле мог записывать и читать такие сообщения. Луи Брайль постарался усовершенствовать эту систему, и через три года (когда ему было пятнадцать) в общих чертах составил собственную, основы которой применяются и сегодня. Много лет такая система использовалась лишь в школах, но постепенно вошла в широкое употребление. В 1835 году Брайль подхватил туберкулез, от которого и умер в 1852 году, в возрасте 43 лет.

Сегодня усовершенствованные варианты системы Брайля соперничают с аудиокнигами, обеспечивая незрячим доступ к письменной информации. Тем не менее шрифт Брайля по-прежнему незаменим и является единственной письменностью, доступной слепоглухим. Шрифт Брайля применяется даже в общественных местах, например в лифтах и банкоматах.

В этой главе мы препарируем код Брайля и разберемся, как он работает. Мы не будем *учить* код Брайля или что-то запоминать. Мы лишь попробуем на этом примере лучше понять его природу.

В шрифте Брайля каждый символ, присутствующий в обычном письменном языке, то есть буквы, цифры и знаки препинания, кодируется в виде одной или нескольких точек в клетке размером две на три точки. Как правило, точки в клетке нумеруются от 1 до 6.



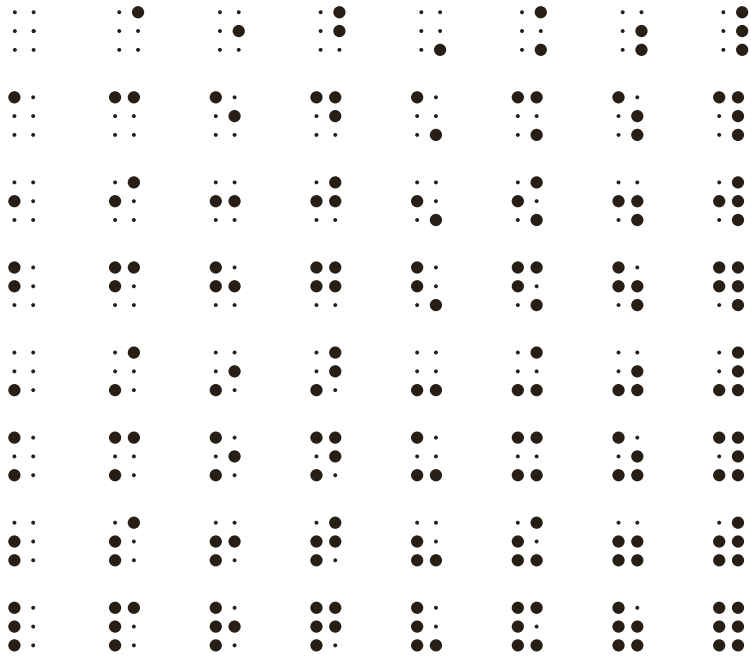
В настоящее время существуют специальные пишущие машинки — брайлевские принтеры, выбивающие точки брайлевского шрифта на бумаге.

Поскольку книга получилась бы запредельно дорогой, если бы хоть пару страниц набрали шрифтом Брайля, я пользовался нотацией, традиционно применяемой для передачи азбуки Брайля при печати. В такой нотации отображаются

все шесть точек в клетке. Жирные точки — это выпуклости на бумаге, мелкие — плоские элементы клетки. Например, в следующем брайлевском символе точки 1, 3 и 5 выпуклые, а 2, 4 и 6 — нет.

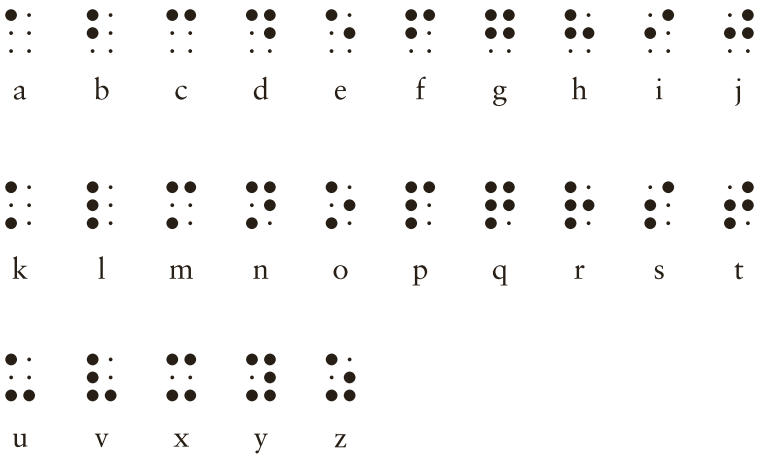
На данный момент нас должно заинтересовать, что эти точки *двоичны*. Любая точка может быть либо выпуклой, либо плоской. Таким образом, шрифт Брайля подчиняется тем же принципам, которые знакомы нам из азбуки Морзе и комбинаторного анализа. Известно, что в клетке шесть точек, и каждая точка может быть плоской или выпуклой, поэтому общее число комбинаций, которые складываются из шести плоских или выпуклых точек, равно $2 \times 2 \times 2 \times 2 \times 2 \times 2$, или 2^6, или 64.

Как видите, в системе Брайля можно представить 64 уникальных кода.



Если в шрифте Брайля используется менее 64 кодов, логично спросить, почему не все возможные варианты в ходу. Если в шрифте Брайля найдется более 64 возможных кодов, значит, сбоит либо наш разум, либо фундаментальные математические истины из разряда «два плюс два равно четырем».

Приступая к изучению шрифта Брайля, рассмотрим, как в нем записываются строчные буквы латиницы.



Например, английская фраза You and me* записывается следующим образом.

Важно: между клетками, соответствующим буквам в слове, ставятся небольшие пробелы; более широкий пробел (в сущности, целая клетка, в которой нет выпуклых точек) соответствует пробелу между словами.

Именно такова основа шрифта Брайля в редакции самого Брайля — как минимум что касается латиницы. Луи Брайль также разработал коды для букв с диакритическими знаками (они часто встречаются во французском языке). Обратите внимание: здесь нет кода для буквы *w*, которая в классическом французском не используется. (Не волнуйтесь, и эта буква вскоре появится.) Пока мы учли всего 25 из 64 возможных кодов.

Внимательно присмотревшись к вышеприведенным строчкам, можно заметить, что в них прослеживается закономерность. В первой строчке (от *a* до *j*) в каждой клетке используются лишь четыре верхние точки: 1, 2, 4 и 5. Второй ряд точно такой же, как первый, но в нем есть и точка 3. Третий ряд подобен первым двум, но в нем мы видим не только точку 3, но и точку 6.

Со времени Луи Брайля его шрифт дополняли различным образом. Современная система, при помощи которой обычно записываются подобные английские тексты, называется «сокращенный Брайль». В сокращенном Брайле много упрощений, помогающих беречь деревья и ускорять чтение. Например, если код некоторой буквы стоит обособленно, то он означает распространенное слово. В следующих трех рядах приведены такие коды для целых слов.

---

* «Ты и я». *Прим. перев.*

(none) (отсутствует)	but но	can может	do делать	every каждый	from от, из	go идти	have иметь	(none)	just только, сейчас

knowledge знание	like как, любить	more больше	not не	(none)	people люди	quite весьма	rather вполне	so так	that так что

us нам, нас, нами	very очень	it оно	you ты	as как, в качестве	and и	for для	of (предлог)	the (артикль)	with с

Таким образом, фразу You and me сокращенным Брайлем можно записать так.

Вот мы и описали 31 код: пробел без точек, который ставится между словами, и три строки по десять кодов, используемых для обозначения букв и слов. Мы до сих пор и близко не израсходовали 64 теоретически доступных кода. Как мы убедимся, в сокращенном Брайле ни один не остался без дела.

Во-первых, можно использовать коды букв *a* — *j*, добавляя к каждому из них выпуклую точку 6. Эти коды применяются в основном для сокращения в слове букв, для буквы *w* и другого сокращения слов.

ch	gh	sh	th	wh	ed	er	ou	ow	w (или «will»*)

** Will — вспомогательный глагол для образования будущего времени.*

Например, слово about* можно записать сокращенным Брайлем вот так.

Во-вторых, можно взять коды букв *a* — *j* и «опустить» их так, чтобы использовались лишь точки 2, 3, 5 и 6. Этими кодами обозначаются некоторые знаки препинания и сокращения, в зависимости от контекста.

ea	bb	cc	dis	en	to к	gg	his его	in в	was был(а)
,	;	:	.		!	()	“		”

Первые четыре приведенных кода — это запятая, точка с запятой, двоеточие и точка. Обратите внимание: как открывающая, так и закрывающая скобки обозначаются одним и тем же кодом, а вот коды для открывающей и закрывающей кавычки отличаются.

Пока мы использовали 51 код. Далее приведены шесть кодов, представляющих различные незадействованные комбинации точек 3, 4, 5 и 6. С их помощью записывают сокращения и некоторые дополнительные знаки препинания.

st /	ing	ble #	ar	'	com -

Код ble очень важен: если это не часть слова, то он означает, что следующие далее коды должны интерпретироваться как числа. Числовые коды точно такие же, как и для букв *a* — *j*.

1	2	3	4	5	6	7	8	9	0

Следовательно, нижеприведенная последовательность означает 256.

* О, около, про. *Прим. перев.*

Если вы следите за нитью повествования, то помните, что до максимума (64) нам остается еще семь кодов. Вот они.

Первый код (выпуклая точка 4) — индикатор ударения. Остальные используются в качестве префиксов при некоторых сокращениях, а также в иных целях. Например, при выпуклых точках 4 и 6 (пятый код в этом ряду) код может означать либо десятичную запятую (для чисел), либо логическое ударение — в зависимости от контекста.

Наконец (если вам не терпится узнать, как в шрифте Брайля записываются заглавные буквы), у нас есть выпуклая точка 6. Это индикатор заглавной буквы. Следующая после такого символа буква будет в верхнем регистре. Например, имя создателя этой системы записывается так.

Здесь индикатор заглавной буквы, буква *l*, буквосочетание *ou*, буквы *i* и *s*, пробел, еще один индикатор заглавной буквы, а далее — буквы *b, r, a, i, l, l* и *e* (на практике эта запись может быть еще короче: отбрасываются две последние буквы, так как они не произносятся).

Итак, мы рассмотрели, как шесть двоичных элементов (точек) дают 64 возможных кода — и не больше. Получается, что многие из этих кодов выполняют двойную работу в зависимости от контекста. Особенно интересны «числовой» и «буквенный» индикаторы (при этом второй отменяет первый). Эти коды меняют семантику других кодов — тех, что следуют за ними: с букв на цифры и обратно с цифр на буквы. Подобные коды часто именуются кодами *старшинства* или *переключения*. Они меняют семантику всех последующих кодов до тех пор, пока переключение не будет отменено.

Индикатор заглавной буквы означает, что следующая (и только следующая) буква должна быть в верхнем, а не в нижнем регистре. Такой код принято называть *экранирующим*, и он «защищает» последовательность других кодов от банальной, рутинной семантики и обеспечивает им новую интерпретацию. Читая следующие главы, убедимся, что коды переключения и экранирующие коды постоянно используются в ситуациях, когда письменный язык нужно представить в двоичном виде.

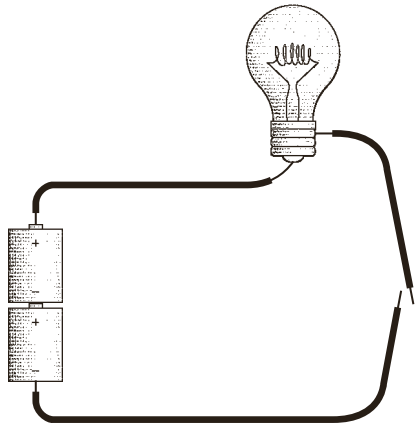
Глава 4
Устройство фонарика

Фонарик многофункционален: чтение под одеялом и обмен зашифрованными сообщениями — лишь два наиболее очевидных варианта его применения. Обычный хозяйственный фонарик может сыграть ключевую роль в наглядном уроке о таком феномене, как электричество.

Электричество — удивительное явление. Сегодня оно используется повсеместно, но при этом окутано тайной даже для тех, кто в нем якобы разбирается. Боюсь, нам так или иначе придется подступиться к этой теме. К счастью, чтобы разобраться, как электричество используется в компьютерах, потребуется понять лишь некоторые базовые концепции, связанные с ним.

Определенно, фонарь — один из простейших электроприборов, имеющийся почти в каждом доме. Можно разобрать обычный фонарик и убедиться, что он состоит из пары батареек, лампочки, выключателя и кое-каких металлических деталей. Все это находится в пластиковом корпусе.

Можно сконструировать заправский светильник, оставив всего две составляющие из этого комплекта: батарейки и лампочку. Кроме того, вам потребуются короткие изолированные проводки (оголенные на кончиках) и умелые руки, чтобы все это держать вместе.

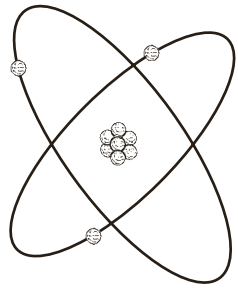


Обратите внимание на два оголенных кончика проводов в правой части схемы. Это наш переключатель. Исходя из того, что батарейки у нас хорошие и лампочка не перегорит, достаточно коснуться одного проводка другим — и загорится свет.

Мы только что сконструировали простую электрическую цепь. Первым делом необходимо отметить, что эта *цепь* представляет собой *круг*. Лампочка зажжется лишь в том случае, если контур от батареек к лампочке, далее к переключателю и обратно к лампочке будет непрерывным. Достаточно любого разрыва — и лампочка погаснет. Выключатель нужен для того, чтобы управлять этим процессом.

Круговая структура подсказывает, что по электрической цепи движется нечто подобное воде, текущей в трубах. Сравнение с водой и трубами довольно распространено при описании сути электричества, но, как и любая аналогия, оно рано или поздно себя исчерпает. Электричество не похоже ни на что иное во Вселенной, его требуется описывать в специфических терминах.

Господствующая научная мудрость, характеризующая природу электричества, называется *электронной теорией*, согласно которой электричество возникает в результате движения электронов.



Известно, что любая материя — вещества, которые можно видеть и осязать, — состоит из крошечных частиц, именуемых атомами. В состав каждого атома входят более мелкие частицы трех типов: нейтроны, протоны и электроны. Атом можно изобразить как миниатюрную Солнечную систему, где нейтроны и протоны связаны в ядре, а электроны вращаются вокруг ядра, как планеты вокруг Солнца.

Необходимо отметить, что вы бы увидели несколько иную картину, будь у вас достаточно мощный микроскоп, позволяющий рассматривать отдельные атомы, но «планетарная» модель довольно удобна.

В атоме, показанном на этой странице, три протона, три электрона и четыре нейтрона, значит, перед нами атом лития. Литий — это один из 118 известных

элементов, каждый из которых обладает собственным *атомным числом* от 1 до 118. Атомное число указывает, сколько протонов в ядре у каждого атома этого элемента, а также сколько электронов в таком атоме. Атомное число лития равно трем.

Атомы могут образовывать химические связи с другими атомами, объединяясь в *молекулы*. Как правило, молекулы обладают совсем иными свойствами, нежели атомы, из которых они состоят. Например, в молекуле воды два атома водорода и один атом кислорода (поэтому химическая формула воды H_2O). Очевидно, вода существенно отличается как от водорода, так и от кислорода. Молекулы поваренной соли состоят из атома натрия и атома хлора, но ни одно из этих веществ не показалось бы вам особо аппетитным, если бы его добавили в картошку фри в чистом виде.

Водород, кислород, хлор, натрий — это всё элементы. Вода и соль — *соединения*. Однако водно-соляной раствор — это смесь, а не соединение, поскольку в растворе вода и соль сохраняют присущие им свойства.

Количество электронов в атоме обычно равно количеству протонов. Случается, что электроны вышибаются из атома. Именно так и возникает электричество.

Слова «электрон» и «электричество» происходят от древнегреческого ηλεκτρον (читается [электрон]).

Может показаться, что это слово означает «крошечная невидимая штука». На самом деле ηλεκτρον в переводе с греческого — «янтарь», прозрачная окаменевшая древесная смола. Такая необычная этимология возникла потому, что древние греки пробовали натирать янтарь шерстяной тканью, а при этом возникает статическое электричество. Когда мы потираем янтарь шерстяной тряпочкой, она вытягивает из камня электроны. В шерсти возникает избыток электронов по сравнению с протонами, а в янтаре электронов становится слишком мало. Более современный подобный эксперимент связан с обычным ковром: если пошаркать по нему, палас захватывает электроны из подошв обуви.

У протонов и электронов есть свойство под названием «*электрический заряд*». Считается, что у протонов положительный заряд (+), а у электронов — отрицательный (–). Нейтроны нейтральны, у них нет заряда. Хотя мы и обозначаем протоны и электроны символами «плюс» и «минус», эти символы в данном случае не имеют арифметической семантики и не означают, что у протонов есть что-то, чего у электронов нет. Противоположные характеристики проявляются именно в том, как протоны и электроны соотносятся друг с другом.

Протоны и электроны наиболее «спокойны» и стабильны, когда в равных количествах сосуществуют рядом. Если возникает дисбаланс между протонами и электронами, он самопроизвольно выправляется. После того как ковер наберет электронов из ваших подошв, ситуация выровняется, стоит вам коснуться его, — проскочит искра. Такая искра статического электричества

возникает в результате движения электронов, которые проделывают практически круговой маршрут — от ковра через все тело, затем опять к подошвам.

Взаимосвязь между протонами и электронами можно описать иначе: противоположные заряды притягиваются, одноименные — отталкиваются. Правда, схема атома производит иное впечатление. Кажется, что протоны сосредоточены в ядре и притягиваются друг к другу. Протоны удерживаются вместе благодаря силе более мощной, чем отталкивание одинаковых зарядов, — *сильному взаимодействию*.

Чтобы подступиться к сильному взаимодействию, требуется расщепить ядро, в результате чего высвободится ядерная энергия. А мы просто балуемся с электронами, чтобы получить электричество.

Статическое электричество — это не просто искорки, проскакивающие, если дотронуться ладонью до дверной ручки. Во время грозы в нижней части тучи накапливаются электроны, а в верхней возникает дефицит электронов; рано или поздно бьет молния, и баланс восстанавливается. Молния — это множество электронов, которые с огромной скоростью летят из одной точки в другую.

Электричество в проводах фонарика, разумеется, гораздо благовоспитаннее, чем в искре или в молнии. Лампочка горит ровно и непрерывно*, поскольку электроны не просто скачут с места на место. Когда один атом в электрической цепи теряет электрон, отдавая его другому атому, он сразу же захватывает электрон от соседнего атома, а тот — от следующего и т. д. Электричество в цепи — это переход электронов от атома к атому.

Все это происходит не само по себе. Нельзя просто взять, соединить проводами всякое барахло и рассчитывать, что в нем потечет электричество. Нужен какой-то инициирующий фактор, который запустит движение электронов в цепи. Возвращаясь к схеме простейшего фонарика, можно предположить, что электричество возникает не в проводах и не в лампочке. По-видимому, источником электричества являются батарейки.

Почти любому известно хотя бы кое-что о типах батареек, используемых в фонариках:

— батарейки цилиндрические бывают разных размеров, например D, C, A, AA, AAA;
— независимо от размера на любой батарейке указана величина 1,5 вольта;
— один кончик батарейки плоский, на нем стоит знак «–»; на другом конце небольшой выступ и знак «+»;

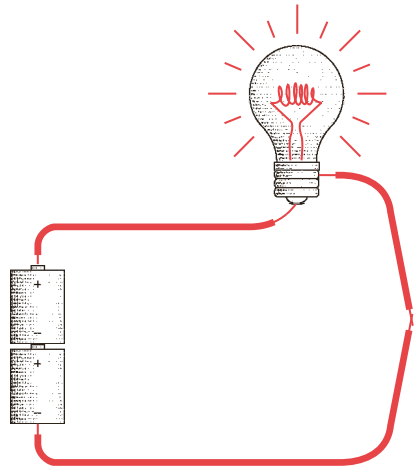
---

* На самом деле интенсивность света лампочки флуктуирует, то есть пульсирует, но наш глаз не способен различать настолько быстрые и мелкие пульсации. *Прим. науч. ред.*

— если вы хотите, чтобы прибор работал нормально, правильно вставляйте батарейки, чтобы плюсы и минусы располагались верно;
— мы полагаем, что батарейки каким-то хитрым образом «дают» электричество.

Во всех батарейках происходят химические реакции: либо одни молекулы распадаются на другие, либо одни молекулы соединяются с другими, образуя третьи. Химические вещества в батарейке подбираются так, чтобы в результате реакции между ними с минусового конца образовывались свободные электроны (этот конец называется «отрицательная клемма» или «катод»), которые так нужны на плюсовом конце батарейки (он же «положительная клемма» или «анод»). Таким образом химическая энергия преобразуется в электрическую.

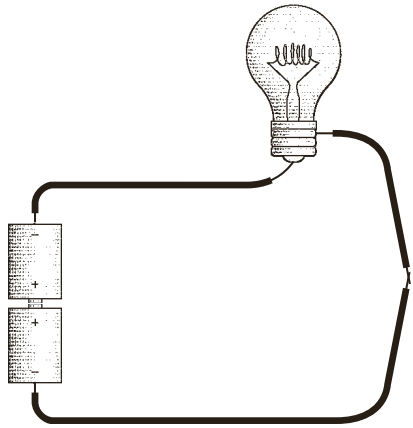
Химическая реакция может протекать лишь при условии, что лишние электроны каким-то образом будут извлекаться с отрицательного полюса батареи и доставляться обратно к положительному. Если батарейка ни к чему не подключена, ничего особенного в ней происходить не будет. (На самом деле химическая реакция там все-таки идет, но очень медленно.) Химические реакции разгоняются лишь при наличии электрического тока, несущего электроны с отрицательного конца батарейки к положительному. Электроны движутся по этой цепи против часовой стрелки*.



В этой книге красным цветом обозначаются провода, по которым течет электричество.

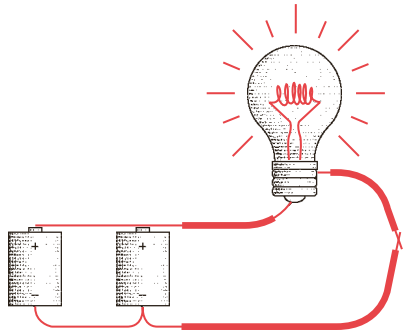
* Принято считать за направление тока направление положительно заряженных частиц. По факту ток создается электронами и течет в противоположном направлении. *Прим. науч. ред.*

Электроны из химикатов, содержащихся в батарейке, могли бы свободно смешиваться с электронами из медного провода, если бы не один простой факт: все электроны, где бы они ни находились, идентичны. Электрон из атома меди ничем не отличается от любого другого электрона.



Обратите внимание: обе батарейки ориентированы в одном и том же направлении. Положительный полюс нижней батарейки принимает электроны с отрицательного полюса верхней батарейки, как будто мы сложили из двух маленьких батареек одну большую, общая мощность которой составляет не 1,5, а 3 вольта.

Если повернуть одну из батареек в противоположную сторону, то электрическая цепь будет разорвана.



Для химических реакций двум положительным полюсам батареи нужны электроны, но поскольку полюса подсоединены друг к другу, путь для электронов закрыт. Если соединены два положительных полюса батарейки, нужно соединить и два отрицательных.

Все работает. Принято говорить, что в таком случае батарейки подключены *параллельно*, а не *последовательно*. Общее напряжение равно 1,5 вольта, как и у каждой батарейки по отдельности. Лампочка горит не слишком ярко, зато батарейки проработают вдвое дольше.

Принято считать, что батарейка подает ток в электрическую цепь. Аналогично можно думать, что электрическая цепь открывает путь для химических реакций в батарейке, которые продолжаются, пока весь химикат в ней не будет израсходован, после чего батарейку нужно выбросить или перезарядить.

Электроны попадают от отрицательного полюса к положительному, проходя через провода и лампочку. Зачем нужны провода? Не может ли электричество передаваться просто по воздуху? И да, и нет. Да, электричество передается по воздуху (особенно если воздух влажный), поэтому и возникают молнии. Но электричество течет по воздуху «неохотно».

Некоторые вещества проводят электричество существенно лучше, чем другие. Это связано со строением атома. Электроны вращаются вокруг ядра на разных энергетических уровнях, которые называются оболочками или орбиталями. Если у атома на внешней оболочке всего один электрон, он легко его отдает, — как раз это и нужно для передачи электричества. Такие вещества хорошо проводят электричество, поэтому их называют *проводниками*. Лучшие проводники — медь, серебро и золото. Не случайно эти элементы расположены в одном и том же столбце периодической системы. Медь — самое распространенное сырье для изготовления проводов.

Свойство, противоположное электропроводимости, называется *сопротивлением*. Некоторые вещества сильнее сопротивляются току, нежели другие, — это *резисторы*. Если вещество обладает очень высоким сопротивлением, практически не проводит электричество, его называют диэлектриком (*изолятором)*. Резина и пластик — хорошие изоляторы, вот почему из них часто делают оболочку для проводов. Ткань и дерево, сухой воздух тоже хороши в таком качестве. Однако при достаточно высоком напряжении практически любой материал приобретает электропроводимость.

Сопротивление меди невелико, но оно *присутствует*. Чем длиннее провод, тем выше его сопротивление. Если бы вы попытались зажечь фонарик с проводами длиной в несколько километров, то их сопротивление оказалось бы чрезмерным, и фонарик бы не работал.

Чем толще провод, тем ниже его сопротивление. Это может показаться нелогичным. Кажется, что чем толще провод, тем больше нужно электричества, чтобы его «наполнить». На самом деле в толстом проводе доступно гораздо больше электронов, образующих электрический ток.

Я уже говорил о напряжении, но не дал определения этому явлению. Напряжение батарейки составляет 1,5 вольта. Что это значит? Напряжение, измеряемое в вольтах (единица напряжения названа так в честь графа Алессандро Вольта (1745–1827), который в 1800 году сконструировал первую батарею), — это одна из самых сложных концепций в элементарной электротехнике. Напряжение описывает *потенциал* для выполнения работы. Напряжение существует независимо от того, подключены ли к батарее какие-либо приборы.

Возьмем, к примеру, кирпич. Когда он лежит на полу, его потенциальная энергия очень мала. Она увеличится, если вы поднимете кирпич на высоту метр двадцать от земли. Чтобы высвободить потенциальную энергию, достаточно отпустить кирпич. Если забраться на крышу высокого здания и поднять кирпич, его потенциальная энергия будет еще больше. Во всех трех случаях вы держите кирпич, сам он ничего не делает, но *потенциал* его отличается.

Гораздо проще определить, что такое *ток*. Сила тока зависит от того, сколько электронов мчится по проводнику. Сила тока измеряется в *амперах*, названных так в честь Андре Ампера (1775–1836). Чтобы достичь силы тока в один ампер, через поперечное сечение проводника нужно пропустить 6 240 000 000 000 000 000 электронов в секунду.

Здесь уместна аналогия с водой, текущей по трубам. Ток подобен *объему* воды, проходящему через трубу в единицу времени, напряжение — *давлению* воды. Сопротивление можно сравнить с шириной трубы: чем уже труба, тем выше сопротивление. Таким образом, чем выше давление, тем больше воды проходит через трубу, чем меньше сечение трубы, тем меньше воды через нее течет. Объем воды, текущей через трубу (ток) в единицу времени прямо пропорционален давлению воды (напряжению) и обратно пропорционален толщине трубы (сопротивлению).

Электротехника позволяет вычислить силу тока, если известны напряжение и сопротивление. Сопротивление — способность вещества тормозить поток электронов — измеряется в омах. Эта единица названа в честь Георга Ома (1789–1854), который также сформулировал знаменитый закон Ома:

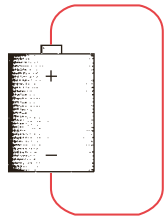
$$I = E / R,$$

где I традиционно обозначает силу тока в амперах, E — *электродвижущая сила, ЭДС* (это первая буква в английском словосочетании electromotive force), а R — сопротивление.

Так, рассмотрим батарею, которая просто лежит в покое и ни к чему не подключена.

ЭДС *E* равна 1,5 вольта. Это потенциал для выполнения работы*. Поскольку между плюсовой и минусовой клеммой лишь воздух, сопротивление получается очень высоким, а значит, сила тока равна 1,5 вольта, деленному на очень большое число. Таким образом, ток практически нулевой.

Теперь соединим положительную и отрицательную клемму коротким отрезком медной проволоки (здесь и далее изоляцию на проводах показывать на рисунках не буду).



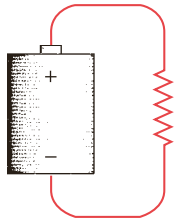
Перед вами *короткое замыкание*. ЭДС по-прежнему равна 1,5 вольта, но сопротивление очень низкое. Узнаем силу тока, разделив 1,5 вольта на очень малое значение. Сила тока получится огромной. По проводу побежит целая уйма электронов. На практике фактическое значение силы тока ограничено физическим размером батареи. Вероятно, батарея просто окажется не в состоянии выдать ток такой силы, и напряжение упадет ниже 1,5 вольта. Если батарея окажется достаточно велика, то провод разогреется, поскольку электрическая энергия станет превращаться в тепловую. Если провод нагреется слишком сильно, он может раскалиться и даже расплавиться.

Большинство электрических цепей попадает в промежуток между этими двумя крайностями. Их можно символически изобразить следующим образом.

Любой электротехник понимает, что зубчатая линия на этом рисунке обозначает резистор. В данном случае показано, что сопротивление в электрической цепи среднее — не высокое, не низкое.

---

* Напряжение — работа поля по переносу заряда. Электрический ток — это изменение объемного заряда во времени, но поскольку со временем заряды (электроны) кончатся, необходим источник, восполняющий «запасы» электронов, которые могли бы перемещаться по проводнику. Таким источником может быть источник ЭДС. ЭДС, в свою очередь, — это работа внешних сил по переносу заряда, то есть сил не электромагнитной природы. В рассматриваемом примере это химические реакции. *Прим. науч. ред.*



Если сопротивление у провода низкое, он может сильно нагреться и раскалиться. Так устроена лампа накаливания. Честь создания электрической лампы накаливания обычно приписывается самому знаменитому американскому изобретателю Томасу Эдисону (1847–1931), но по состоянию на 1879 год, когда он запатентовал электролампочку, принцип ее работы был хорошо известен, и другие ученые тоже работали над этой проблемой*.

Внутри лампы находится тонкая проволока, именуемая «нить накаливания», которая обычно изготавливается из вольфрама. Один кончик этой спирали подключен к нижнему контакту металлического цоколя, другой — к резьбовой поверхности цоколя, причем между нижним контактом и резьбой цоколя проложен изолятор. Провод обладает сопротивлением, поэтому нагревается. На воздухе вольфрамовая спираль раскалилась бы настолько, что просто сгорела бы, но внутри лампочки вакуум, поэтому раскаленная нить накаливания хорошо светится.

В типичном фонарике — две батарейки с последовательным соединением. Общее напряжение составляет три вольта. Сопротивление типичной лампочки из карманного фонарика — четыре ома. Следовательно, чтобы узнать силу тока в такой лампочке, делим три вольта на четыре ома и получаем 0,75 ампера, или 750 миллиампер. Таким образом, каждую секунду через лампочку пролетает 4 680 000 000 000 000 000 электронов.

Краткая проверка на практике: если попытаться измерить сопротивление лампочки карманного фонарика при помощи омметра, результат получится гораздо ниже четырех омов. Сопротивление вольфрама зависит от температуры, и по мере нагревания лампочки оно возрастает.

Вероятно, вы знаете, что на бытовых лампочках пишут, сколько в них ватт. Эта единица названа в честь Джеймса Уатта (1736–1819), прославившегося своей работой над паровым двигателем. Ватт — это единица мощности (P), которая вычисляется по формуле:

---

* В России разработкой ламп накаливания занимался физик Александр Николаевич Лодыгин. *Прим. науч. ред.*

$$P = E \times I.$$

Показатели нашего фонарика — три вольта и 0,75 ампера, то есть мы имеем дело с лампочкой мощностью 2,25 ватта.

Возможно, у вас в комнате горит стоваттная лампочка, которая рассчитана на бытовое напряжение 120 вольт. Следовательно, сила тока, идущего через такую лампочку, равна 100 ватт разделить на 120 вольт, то есть примерно 0,83 ампера. Таким образом, сопротивление стоваттной лампы накаливания равно 120 вольт разделить на 0,83 ампера — примерно 144 ома.

Кажется, мы проанализировали все элементы фонарика: батарейки, провода, лампочку. Но забыли о самом важном!

Да, еще выключатель. От положения выключателя зависит, есть ли ток в электрической цепи. Когда ток идет, говорят, что фонарик *включен*, или *контур замкнут*. Когда фонарик *выключен* (*контур разомкнут*), ток идти не может. Таким образом, провод и дверь в некотором смысле противоположны: когда дверь закрыта (замкнута), через нее нельзя пройти, а в случае с проводом всё наоборот.

Выключатель либо включен, либо выключен, ток или идет, или нет, лампочка или светится, или не светится. Подобно двоичным кодам, изобретенным Морзе и Брайлем, обычный фонарик может быть лишь в двух состояниях: включен либо выключен. Промежуточных состояний не существует. Понимание сходства между двоичными кодами и простыми электрическими цепями нам еще пригодится.

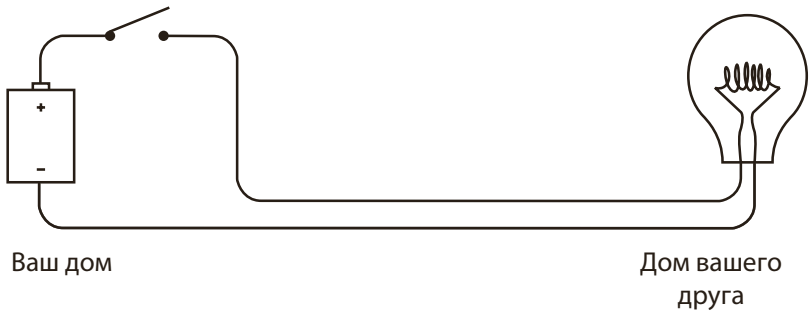
Глава 5
Заглядывая за угол

Вам двенадцать. Наступает ужасный день: семья вашего лучшего друга переезжает в другой город. Вы время от времени перезваниваетесь, но разве сравнишь беседы по телефону с полуночными посиделками, когда вы, вооружившись фонариками, сигнализировали друг другу азбукой Морзе! В итоге вы близко сходитесь еще с одним другом, живущим по соседству. Теперь надо обучить его азбуке Морзе, чтобы общаться за полночь, обмениваясь фонарными вспышками.

Проблема в том, что окно вашей спальни и окно спальни нового друга не обращены друг к другу. Дома стоят на одной улице, но окна смотрят в одну и ту же сторону. Если на улице не получится каким-то образом установить систему зеркал, азбукой Морзе через окно не пообщаешься.

Или все же пообщаешься?

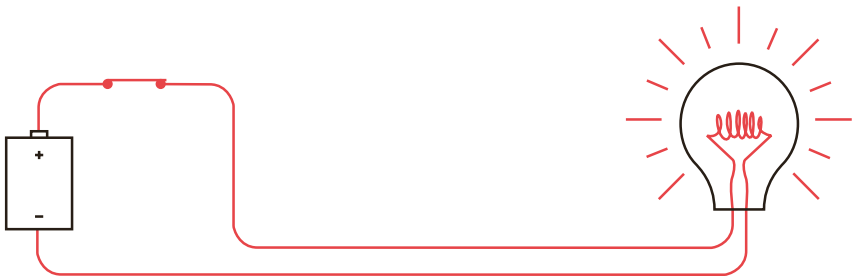
Вероятно, к тому моменту вы уже что-то узнали об электричестве, так что решаете собрать собственные фонарики из батареек, лампочек, выключателей и проводов. Первым делом вы прямо в спальне соединяете батарейки и выключатель. Два провода тянутся из окна через забор в спальню вашего друга, где он подключает их к лампочке.



Я показываю всего одну батарейку, но вы можете пользоваться двумя. Здесь и далее на схемах так будет обозначаться выключенный (разомкнутый) переключатель.

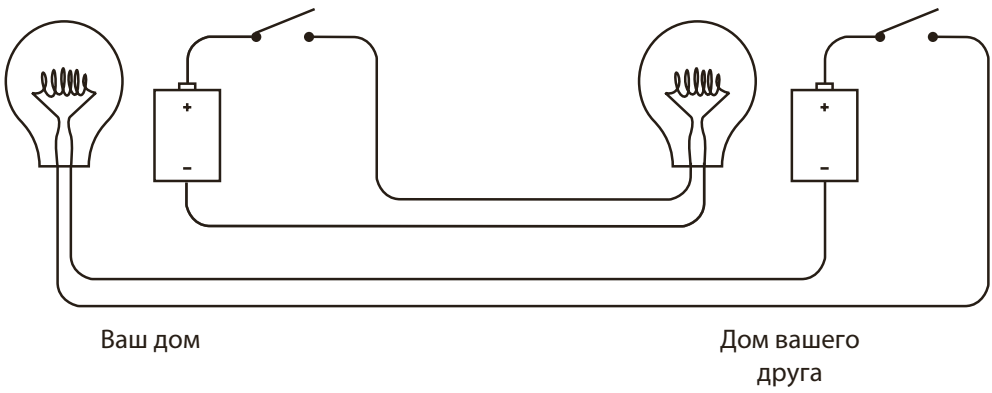
А так — включенный (замкнутый).

Фонарик в этой главе работает по тому же принципу, что и в предыдущей, но провода, подключаемые к элементам схемы, немного длиннее. Когда вы замыкаете цепь, лампочка зажигается в комнате вашего друга.



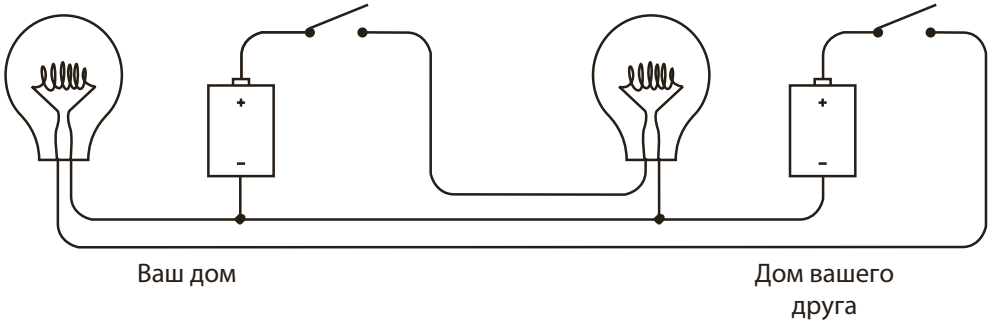
Теперь вы можете обмениваться сообщениями при помощи азбуки Морзе.

У вас получился один «дальнобойный» фонарик; значит, можно подключить второй, которым будет пользоваться ваш друг.



Поздравляем! Вы только что соорудили двунаправленный телеграф. Как видите, здесь две одинаковые электрические цепи, которые совершенно не зависят друг от друга и нигде одна с другой не соединяются. Теоретически вы можете отправлять сообщение в момент, когда друг отправляет его вам (хотя это серьезная умственная нагрузка — отправлять и читать сообщения одновременно).

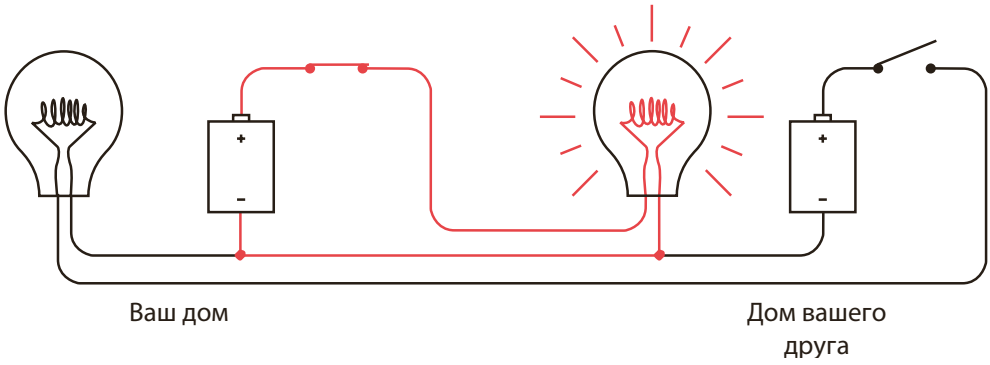
Возможно, вы догадаетесь, что длину проводов можно сократить на четверть, выстроив такую конфигурацию.



Обратите внимание: теперь мы соединили отрицательные клеммы двух батареек. Две кольцевые электрические цепи (от батарейки к выключателю, от выключателя к лампочке и от лампочки к батарейке) по-прежнему работают независимо друг от друга, хотя они и соединены, подобно сиамским близнецам.

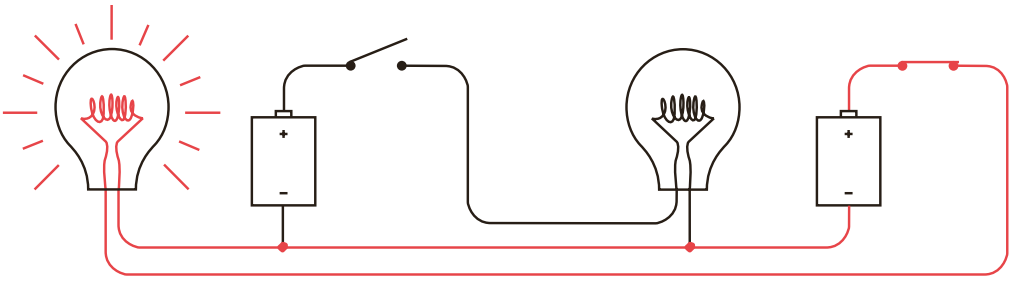
Такое соединение называется «*с общим проводом*». В этой схеме общий провод проложен от левой оконечности, где соединены левая лампочка и батарейка, до правой, где соединены правая лампочка и батарейка. Эти подключения обозначены точками.

Давайте убедимся, что никаких фокусов тут нет. Во-первых, если нажать переключатель на вашей стороне, загорится лампочка дома у вашего друга. Красными линиями показано направление тока в электрической цепи.

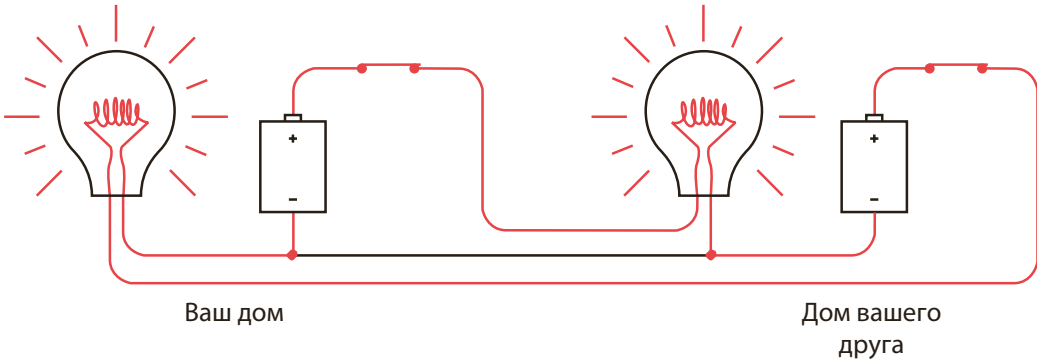


В другую часть схемы электричество не попадает: электронам туда попросту не добраться.

Когда сигнал отправляете не вы, а ваш друг, лампочка у вас в комнате зажигается и гаснет от выключателя, находящегося у него в спальне. Опять же, направление электричества в цепи показано красными линиями.



Когда вы одновременно с другом пытаетесь передать сигналы, в некоторые моменты оба переключателя выключены, в других случаях один включен, а второй выключен, в третьих — оба включены. Тогда направление тока в цепи выглядит так.



По общему проводу ток не идет.

Когда мы соединяем две цепи в одну при помощи общего провода, у нас остается три провода вместо четырех, и длина всей проводки уменьшается на 25%.

Если бы нам пришлось протянуть провода на достаточно большое расстояние, возможно, мы захотели бы сэкономить и избавиться еще от одного провода. К сожалению, это невозможно при работе с 1,5-вольтными батарейками и маленькими лампочками. Однако если вооружиться стовольтными батареями и более крупными лампами, вероятно, все получится.

Вот какой фокус: как только вы оборудовали общую часть цепи, на этом отрезке уже необязательно использовать провод; его можно заменить чем-нибудь

еще. Например, шаром диаметром 12 тысяч километров, состоящим из металла, камней, воды и органических веществ. Этот гигантский шар — планета Земля.

В прошлой главе, описывая хорошие проводники, я упоминал серебро, медь и золото, но ничего не сказал о гальке и перегное. Земля и правда не идеальный проводник, хотя некоторые виды грунта (например, влажная почва) проводят электричество лучше других (в частности, сухого песка). Существует одно общее правило, касающееся проводников: чем он больше, тем лучше. Очень толстый провод обеспечивает большую электропроводимость, чем очень тонкий. Вот в чем главное достоинство Земли: она огромная.

Чтобы задействовать Землю в качестве проводника, мало просто воткнуть провод в грядку с помидорами. Нужно устройство, которое обеспечит хороший контакт, — я имею в виду, что у проводника должна быть обширная поверхность. В данном случае хорошо подойдет медный прут длиной хотя бы 2,5 метра и примерно 1,5 сантиметра в диаметре. Тогда мы получим площадь контакта проводника с землей, равную 1200 $см^2$. Такой прут можно загнать в землю кувалдой, а затем подключить к нему провод. Если у вас дома проложены медные водопроводные трубы, выходящие из земли где-то за домом, провод можно подсоединить к подобной трубе.

Термин «*заземление*» немного неудачный, поскольку именно им обозначается и тот элемент цепи, который мы выше назвали *общим проводом*. В этой главе (если не будет указано иное) «заземление» означает физическое соединение с грунтом.

На схемах электрических цепей Земля обозначается так.

Электрики пользуются таким символом, потому что им лень рисовать саженный медный прут, закопанный в землю.

Рассмотрим, как все устроено. В начале главы была приведена вот такая однонаправленная конфигурация.

Если работать с достаточно мощными лампочками и батарейками, между вашим домом и домом вашего друга потребуется протянуть всего один провод, ведь в качестве второго проводника будет использоваться Земля.

Когда вы включите систему, электричество потечет так.

Электроны попадают в дом вашего друга прямо из земли, проходят через лампочку и провод, через выключатель у вас дома, а затем отправляются на положительную клемму батарейки. Электроны с отрицательной клеммы батарейки идут в землю.

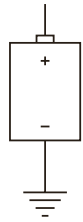
Возможно, вы также пожелаете изобразить электроны, вылетающие из саженного медного прута, закопанного на заднем дворе вашего друга.

Если учесть, что Земля в данном случае выполняет точно такую же функцию для тысяч электрических цепей по всему миру, возможен вопрос: откуда электроны знают, куда именно лететь? Разумеется, не знают. В данном случае удобнее описать Землю при помощи другой метафоры.

Да, Земля — огромный проводник, но ее можно рассматривать и как хранилище, и как источник электронов. *Земля полна электронами, как океан — каплями воды*. Земля — не только неисчерпаемый источник электронов, но и огромный «сток» для этих частиц.

Однако Земля обладает *некоторым* сопротивлением. Вот почему не применяется заземление, когда требуется укоротить провода при опытах с батареями и сигнальными лампочками. Сопротивление Земли просто слишком велико, если речь идет о работе с низковольтными батарейками.

Обратите внимание: на двух предыдущих схемах батарейка заземлена через отрицательную клемму.



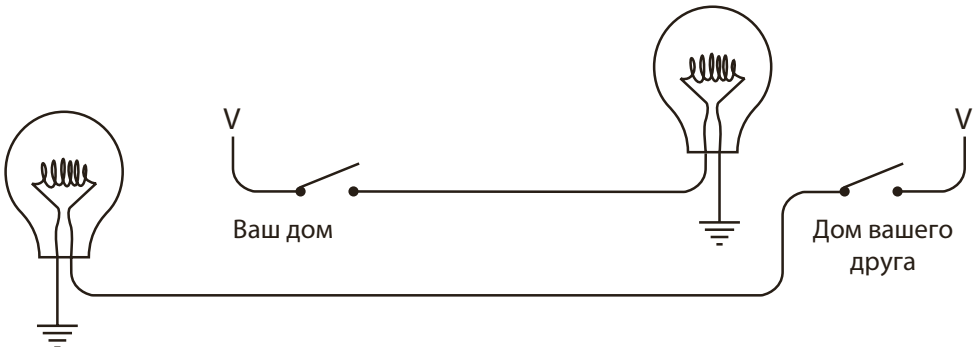
Я больше не буду рисовать заземленную батарейку. Вместо этого стану писать заглавную букву *V*, которая означает напряжение. Теперь однонаправленный телеграф с лампочкой выглядит так.

V означает «напряжение» и «вакуум». Считайте, что V — это электронный вакуум, а Земля — океан электронов. Электронный вакуум тянет электроны из Земли через электрическую цепь, тем временем совершая работу (например, зажигая лампочку).

Точка заземления иногда именуется *точкой с нулевым потенциалом*. Это значит, что в ней отсутствует напряжение. Как я уже рассказывал, напряжение — это потенциал для выполнения работы, и приводил пример с кирпичом, поднятым в воздух и обладающим потенциальной энергией. Нулевой потенциал будет у кирпича, лежащего на земле: оттуда некуда падать.

В главе 4 мы отметили, что электрические цепи закольцованы. Наша новая цепь совершенно не похожа на кольцо. Однако она все равно закольцована. Можно заменить V на батарейку, заземленную через отрицательную клемму, а затем нарисовать провод между всеми точками, где стоит символ заземления. Получится такая же схема, как и приведенная в начале этой главы.

Итак, вооружившись парой медных штырей (или водопроводных труб), можно сконструировать двунаправленную систему для обмена кодом Морзе и при этом обойтись всего двумя проводами, которые будут протянуты через изгороди между вашим домом и домом вашего друга.



Функционально эта цепь не отличается от конфигурации, показанной выше, где через забор между двумя домами протянуты три провода.

Итак, мы рассмотрели важный этап в развитии телекоммуникаций. Ранее мы могли общаться при помощи азбуки Морзе, но только по прямой, в пределах видимости, и только на таком расстоянии, на какое добивает луч фонарика.

При помощи проводов мы изготовили систему, которая позволяет не только общаться с другом «по кривой» (вне зоны прямой видимости), но и избавиться от ограничений, связанных с расстоянием между нами. Можно общаться с кем-то, до кого сотни и тысячи километров, — нужно лишь протянуть достаточно длинные провода.

Нет, в принципе, не совсем так. Пусть медь и очень хороший проводник, она неидеальна. Чем длиннее провода, тем выше их сопротивление. Чем выше сопротивление, тем слабее проходящий по ним ток, чем слабее ток — тем тусклее светит лампочка.

Итак, насколько длинные провода мы можем протянуть? Зависит от ситуации. Допустим, мы работаем с исходной двунаправленной конструкцией на четыре провода, без заземления и общего провода, используем батарейки от фонарика, а также лампочки. Можно для начала приобрести акустический кабель 20-го калибра. Такой кабель обычно применяется для подключения микрофона к стереосистеме. В нем два провода, так что он хорошо подойдет и для двунаправленного телеграфа. Если между вашей комнатой и комнатой друга меньше 15 метров, потребуется всего одна катушка провода.

Толщина провода измеряется по системе AWG (American Wire Gauge, американский калибр проводов)*. Чем меньше калибр, тем толще провод, соответственно тем ниже его сопротивление. Диаметр провода 20-го калибра — около

* В России толщина провода традиционно описывается через площадь сечения, которая измеряется в квадратных миллиметрах, либо через диаметр провода в миллиметрах. *Прим. науч. ред.*

0,8 миллиметра, а сопротивление — 10 омов на 300 метров либо один ом на удвоенное расстояние между комнатами.

Неплохо, но что делать, если бы мы протянули провод на полтора километра? Общее сопротивление такого провода составило бы более 100 омов. Как вы помните, сопротивление нашей лампочки составляло всего четыре ома. По закону Ома можно рассчитать, что сила тока, который потечет по такой цепи, составит уже не 0,75 ампера (три вольта, деленные на четыре ома), а менее 0,03 ампера (три вольта, деленные более чем на 100 омов). Наверняка лампочка от такого низкого тока не загорится. Хороший выход — взять провод потолще. Но это может выйти дороже. Провода 10-го калибра потребуется вдвое больше, поскольку он одножильный, толщина его составляет около 2,54 миллиметра, но сопротивление — всего около пяти омов на 1,6 километра. Другое решение — увеличить напряжение и взять лампочки с гораздо более высоким сопротивлением. Например, стоваттная лампочка, освещающая вашу комнату, рассчитана на работу в сети напряжением 120 вольт и имеет сопротивление около 144 омов. В таком случае сопротивление проводов в меньшей степени отразится на всей нашей схеме.

Именно с такими проблемами столкнулись инженеры, которые 150 лет назад прокладывали первые телеграфные системы между Америкой и Европой. Независимо от толщины проводов и уровня напряжения, телеграфный провод просто невозможно протянуть на неограниченное расстояние. Согласно имевшейся схеме, работоспособная система могла охватить максимум несколько сотен километров, что несравнимо меньше тех тысяч километров, которые пролегают между Нью-Йорком и Калифорнией.

Решить проблему удалось, отказавшись от лампочек и сконструировав простые «щелкающие» телеграфы прошлого века. Получилось простое и неброское устройство, на основе которого впоследствии были созданы полноценные компьютеры.

Глава 6
Телеграфы и реле

Сэмюэл Морзе родился в 1791 году в городе Чарльстауне. Сейчас это северо-восточная часть Бостона. К моменту рождения Морзе минуло уже два года, как ратифицировали Конституцию США. Шел первый президентский срок Джорджа Вашингтона, в России правила Екатерина Великая. Людовик XVI и Мария-Антуанетта спустя два года будут обезглавлены во время Французской революции. В 1791 году Моцарт завершил свою последнюю оперу «Волшебная флейта» и в тот же год умер в возрасте 35 лет.

Морзе получил образование в Йеле и изучал искусство в Лондоне. Он стал успешным портретистом. Портрет генерала Лафайета (1825) кисти Морзе до сих пор экспонируется в Ратуше Нью-Йорка. В 1836 году Морзе баллотировался в мэры Нью-Йорка как независимый кандидат и получил 5,7% голосов. Кроме того, он был одним из первых, кто всерьез увлекался фотографией. Морзе учился у самого Луи Дагера и сделал одни из первых дагеротипов в Америке. В 1840 году он обучил этому искусству 17-летнего Мэтью Брэди, который вместе с коллегами впоследствии создал один из самых запоминающихся снимков Гражданской войны в США, портреты Авраама Линкольна и Сэмюэла Морзе. Все это лишь ремарки к его разносторонней карьере. В наши дни Сэмюэл Морзе наиболее известен как изобретатель телеграфа и азбуки, названной в его честь.

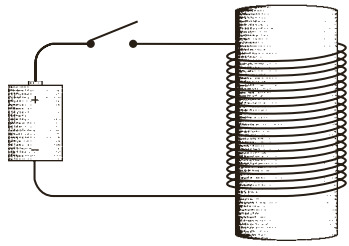
Мгновенная связь в глобальных масштабах, к которой мы так привыкли, — относительно недавнее изобретение. В начале XIX века можно было общаться либо в реальном времени, либо дистанционно, но то и другое сразу было невозможно. Дистанция реального общения была ограничена силой голоса (никаких звукоусилителей не существовало) и зоркостью собеседника (правда, вас могли рассматривать в подзорную трубу). Общаться на больших расстояниях можно было по переписке; для доставки писем требовались время и транспорт: лошади, поезда или корабли.

За многие десятилетия до изобретения, сделанного Морзе, предпринимались многочисленные попытки ускорить дистанционную коммуникацию. Самые примитивные варианты были связаны с выстраиванием цепочек

людей-передатчиков. Они стояли на холмах и размахивали флажками, пользуясь семафорной азбукой. Существовали и более сложные конструкции с руками-манипуляторами, которые, в сущности, выполняли те же функции, что и люди-семафоры.

Идея телеграфа (в буквальном переводе с греческого «пишу далеко») в начале XIX века определенно витала в воздухе, и кроме Морзе за нее пытались браться другие изобретатели. Морзе приступил к экспериментам в 1832 году. В принципе, идея электрического телеграфа проста: на одном конце провода проделываем какие-то манипуляции, эффект которых наблюдается на другом конце. Именно это и получилось у нас, когда мы конструировали дальнобойный фонарик. Однако Морзе не мог пользоваться лампочкой в качестве сигнального устройства, поскольку саму лампочку изобрели лишь в 1879 году. Вместо этого он задействовал явление *электромагнетизма*.

Если взять железный прут, обмотать его несколькими сотнями петель тонкого провода, а затем пропустить по этому проводу ток, прут превратится в магнит. Тогда он станет притягивать другие железные и стальные предметы. (В электромагните хватает тонкого провода, чтобы возникало достаточно высокое сопротивление, не допускающее короткого замыкания). Если отрубить ток, то железный прут теряет магнитные свойства.



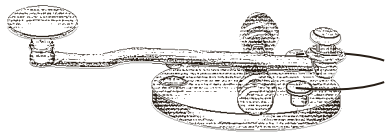
Электромагнит — основа телеграфа. Когда мы включаем или выключаем рычаг с одной стороны цепи, эффект наблюдается на другой стороне.

Первые телеграфы Морзе были сложнее более поздних моделей. Морзе считал, что телеграф должен выводить какую-то информацию на бумаге, как потом будут говорить компьютерщики, создавать физическую копию. Естественно, это не обязательно должны быть слова, поскольку это слишком сложно. Но *что-то* на бумаге нужно записывать, будь то каракули или точки и тире. Обратите внимание: Морзе не мог выйти из плоскости и думал о бумаге и чтении, точно как Валентин Гаюи полагал, что в книгах для слепых должны быть выпуклые буквы алфавита.

Хотя Сэмюэл Морзе уже в 1836 году уведомил патентное бюро о том, что изобрел рабочую модель телеграфа, лишь в 1843 году ему удалось

добиться разрешения на публичную демонстрацию этого устройства в Конгрессе. Исторический день наступил 24 мая 1844 года, когда телеграфная линия связала Вашингтон и Балтимор и по телеграфу удалось успешно передать библейскую фразу: «Чудны дела Твои, Господи».

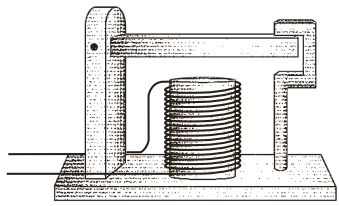
Обычный телеграфный «ключ» для передачи ссобщений выглядел примерно так:



Несмотря на вычурный вид, это был просто переключатель, оптимизированный для максимально скоростной работы. Чтобы подолгу работать с таким ключом, его было удобнее удерживать между большим, указательным и средним пальцами и стучать им вверх-вниз. Короткий удар ключом соответствовал точке из азбуки Морзе, длительное нажатие — тире.

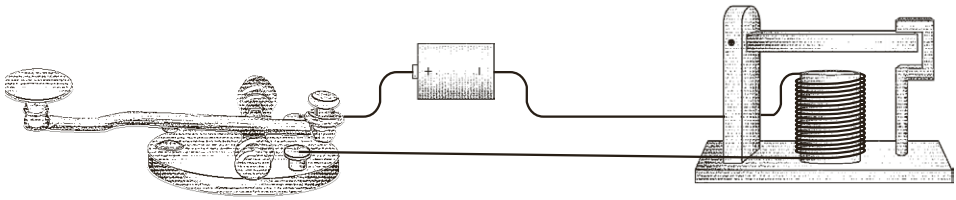
С другой стороны цепи располагался приемник, представлявший собой электромагнит, управлявший металлическим рычагом (изначально электромагнит управлял пером). Пока механизм, оснащенный натянутой пружиной, медленно протягивал бумажный свиток через устройство, перо скакало по бумаге, выписывая на ней точки и тире. Человек, умеющий читать азбуку Морзе, переводил эти точки и тире в буквы и складывал слова.

Да, люди ленивы, и телеграфисты вскоре обнаружили, что код вполне можно переводить, прислушиваясь к длительности ударов пера. В итоге от пера отказались, заменив его более традиционным телеграфным клопфером, который выглядел примерно так.



При нажатии телеграфного ключа электромагнит в клопфере опускал подвижную планку и делал характерный «клик». Когда ключ отпускали, планка отскакивала обратно и издавала звук «клак». Быстрое «клик-клак» соответствовало точке, более долгое «клик-клак» — тире.

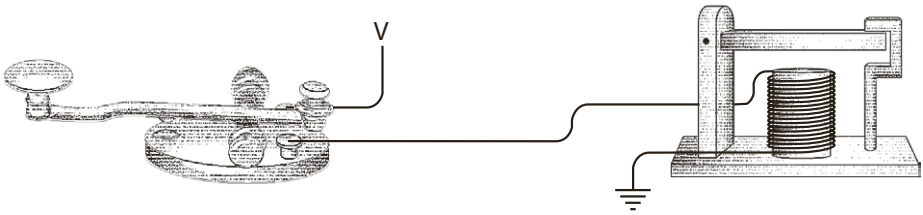
Ключ, клопфер, батарею и несколько проводов можно подключить друг к другу, как в случае со световым телеграфом, о котором мы говорили в предыдущей главе.



Ваша телеграфная станция | Телеграфная станция вашего друга

Для соединения двух телеграфных станций достаточно одного провода, вторую часть цепи замкнем через землю.

Как и в прошлой главе, заменим подключенную к земле батарею буквой *V*. Соответственно, полноценное однонаправленное устройство будет выглядеть так.



Ваша телеграфная станция | Телеграфная станция вашего друга

Для двунаправленной связи нам просто потребуются еще один ключ и передатчик. Примерно такое устройство мы и собирали.

Именно с изобретения телеграфа начинается эпоха современных телекоммуникаций. Впервые людям удалось общаться с собеседником за пределами видимости и слышимости, причем гораздо оперативнее, чем при отправке почты на галопирующей лошади. Гораздо интереснее, что в этом изобретении применялся двоичный код. В более современных средствах кабельной и беспроводной телекоммуникации (телефон, радио, телевизор) от двоичного кода отказались, и вновь он вошел в употребление с возникновением компьютеров, компакт-дисков, цифровых видеодисков, цифрового спутникового телевещания и телевидения высокого разрешения.

Телеграф Морзе превзошел другие модели отчасти потому, что хорошо работал при помехах на линии. Как правило, провод между ключом и клопфером оставался функционален. Другие телеграфные системы были не столь

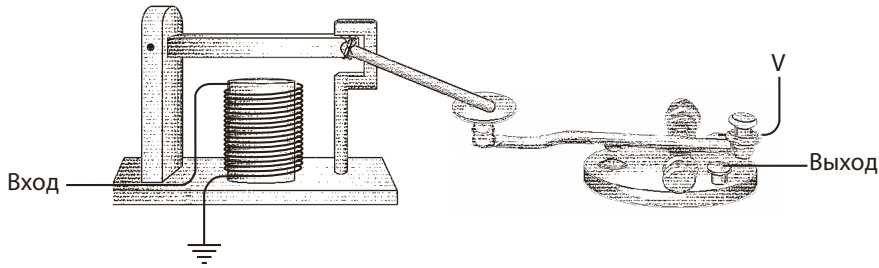
неприхотливы. Я уже упоминал, что большая техническая проблема, связанная с телеграфом, заключается в сопротивлении длинных проводов. Хотя на некоторых телеграфных линиях использовалось напряжение до 300 вольт, и они нормально работали на расстоянии до 480 километров, неограниченно длинных проводов не бывает.

Решение сконструировать систему ретрансляторов очевидно. Через каждые 320 километров можно усадить оператора, дать ему ключ и клопфер и поручить: «Получил сообщение — передай его следующему».

Теперь представьте, что телеграфная компания пригласила вас на работу в качестве такого оператора. Посадили вас где-нибудь в глуши между Нью-Йорком и Калифорнией в хижине, где есть только стол и стул. Через восточное окно в комнату протянут провод, подключенный к клопферу. Телеграфный ключ запитан от батареи, а из батареи в западное окно протянут второй провод. Ваша задача — принимать входящие сообщения из Нью-Йорка и пересылать их в Калифорнию.

Поначалу вы предпочитаете дождаться целостного сообщения, а затем переслать его. Записываете буквы, соответствующие щелчкам клопфера, а когда сообщение закончится — пересылаете, отстукивая ключом. Рано или поздно вы догадаетесь, что сообщение удобнее транслировать прямо в процессе получения, не записывая его целиком. Так экономится время.

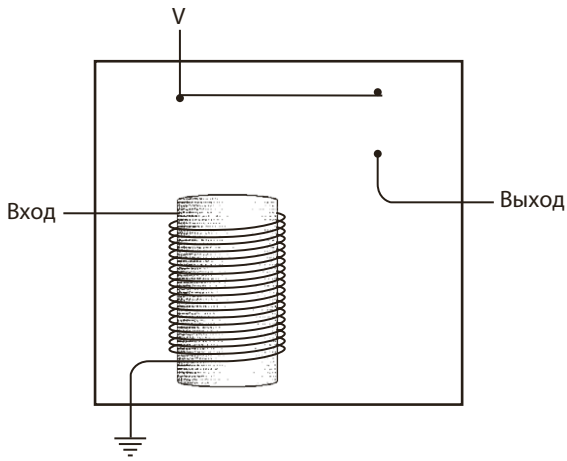
Однажды вы пересылаете сообщение, смотрите, как скачет вверх-вниз планка клопфера, смотрите на собственные пальцы, как вы управляетесь с ключом. Снова смотрите на клопфер, снова на ключ — и осознаете, что ключ скачет в унисон с клопфером. Выходите на улицу, берете дощечку, находите шнурок и при помощи дощечки и шнурка связываете клопфер с ключом.



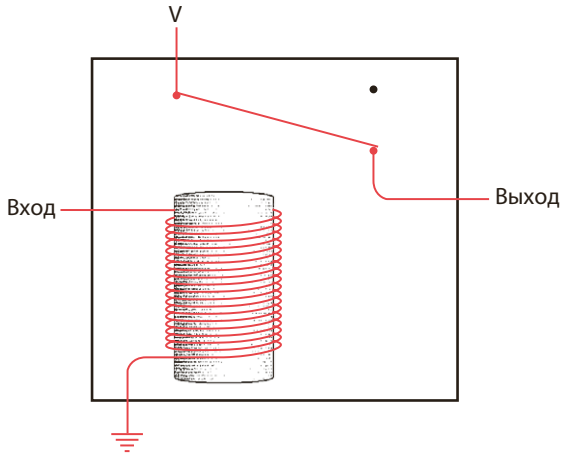
Теперь все работает само, а вы можете устроить свободный вечер и пойти порыбачить.

Интересно нафантазировано. В действительности Сэмюэл Морзе еще на самом раннем этапе концептуально представлял себе такое устройство. Мы изобрели устройство под названием *повторитель*, или *реле*. Реле напоминает клопфер, где входящий ток запитывает магнит, тянущий металлический рычаг. Однако рычаг — это элемент переключателя, соединяющего батарею с исходящим проводом. В таком случае слабый входящий ток «усиливается», и исходящий ток получается гораздо значительнее.

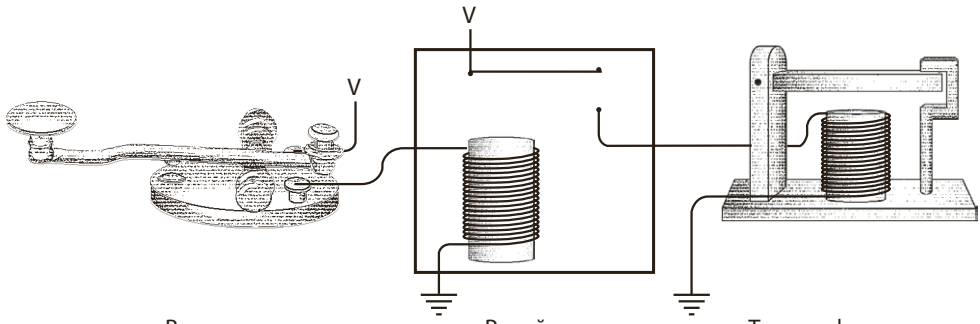
Схема реле выглядит так.



Когда входящий ток активирует электромагнит, последний подтягивает гибкую металлическую ленту, действующую как переключатель, пускающий исходящий ток.



Итак, телеграфный ключ, клопфер и реле соединяются примерно следующим образом.



Реле — замечательное устройство. Безусловно, это переключатель, но такой, который переводится из состояния «включен» в состояние «выключен» и обратно не человеческой рукой, а силой тока. При помощи такого прибора можно делать удивительные вещи, а телеграф в существенной степени позволяет смоделировать компьютер.

Да, реле слишком аппетитное изобретение, чтобы просто оставить его пылиться в музее связи. Заходим в музей, хватаем его, засовываем во внутренний карман пиджака и быстро ретируемся. Реле нам пригодится. Однако прежде чем приступить к работе с ним, нужно научиться считать.

Глава 7
Наши десять цифр

Идея, что язык — просто код, вполне логична. Многие как минимум пытались выучить иностранный язык в старших классах, поэтому сложно поспорить, что кошка в других языках может называться cat, gato, chat, Katze, kot или κατα.

Кажется, что числа менее пластичны в культурном контексте. Независимо от того, на каком языке мы говорим, как произносим числительные, практически любой собеседник на этой планете, скорее всего, будет записывать числа точно так, как и мы.

1 2 3 4 5 6 7 8 9 10

Не потому ли математику называют универсальным языком?

Несомненно, числа — самый абстрактный код, с которым приходится иметь дело в повседневной жизни. Видя число, мы не пытаемся его мгновенно с чем-то соотнести.

3

Можно представить три яблока или три других предмета, но с тем же успехом можно узнать из контекста, что речь идет о дне рождения ребенка, телевизионном канале, хоккейном счете, количестве чашек муки, нужных для приготовления пирога. Уже потому, что числа столь абстрактны, нам сложнее понять, что три яблока можно обозначить не только числом 3.

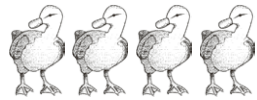
Бóльшая часть этой главы и вся следующая помогут убедиться, что ровно такое же количество яблок можно обозначить и числом 11.

Для начала давайте расстанемся с мыслью, что в числе 10 есть нечто особенное. Неудивительно, что в большинстве цивилизаций сложились системы счисления на основе числа 10 (или 5). С глубокой древности люди считали на пальцах. Если бы у человеческой особи было восемь или двенадцать пальцев, то все счетные системы были бы немного иными.

Именно поэтому система счисления *с основанием 10*, также именуемая *десятеричной*, выбрана совершенно произвольно. Мы придаем десятке чисел поистине магическое значение и придумали для нее особые названия. Десять дней образуют декаду, десять десятилетий — век, десять веков — тысячелетие. Тысяча тысяч — это миллион, тысяча миллионов — миллиард. Все эти числа являются степенями числа 10.

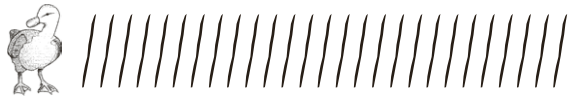
$10^1 = 10$
$10^2 = 100$
$10^3 = 1000$ (тысяча)
$10^4 = 10\,000$
$10^5 = 100\,000$
$10^6 = 1\,000\,000$ (миллион)
$10^7 = 10\,000\,000$
$10^8 = 100\,000\,000$
$10^9 = 1\,000\,000\,000$ (миллиард)

Большинство историков полагают, что числа изначально были придуманы для подсчета предметов, например людей, имущества и торговых сделок. Если у кого-то было четыре утки, то их можно было обозначить в виде четырех нарисованных уточек.



Наконец человек, чья работа заключалась в рисовании уток, подумал: «Зачем рисовать четырех уток? Не изобразить ли одну и обозначить, что на самом деле уток четыре, скажем, черточками?»

Когда потребовалось нарисовать 27 уток, черточки выглядели нелепо.



Подумалось: «Должен быть другой способ, лучше», — так появилась система чисел.

Из всех древнейших числовых систем до сих пор в ходу римские цифры. Они встречаются на циферблатах, ими выбивают даты на памятниках, нумеруют некоторые страницы в книгах, используют при подсчете некоторых элементов и — что наиболее раздражает — при указании информации об авторских правах в кинофильмах. (Иногда чтобы ответить на вопрос, в каком году был снят фильм, нужно молниеносно расшифровать какие-нибудь MCMLIII в хвосте титров.)

Двадцать семь уток римскими цифрами будет так.

Принцип довольно прост: *X* означает 10 черточек, *V* — пять.

Вот римские цифры, сохранившиеся до наших дней.

I V X L C D M

I — это единица; она похожа на черточку или один поднятый палец. *V* — это пятерка; возможно, этим символом обозначалась ладонь. Из двух *V* составляется *X*, то есть десятка.

L — это пятьдесят, *C* — буква, с которой начинается латинское centum, — сто, *D* — пять сотен, *M* — первая буква в слове mille — тысяча.

Хотя мы, возможно, с этим не согласимся, но на протяжении веков считалось, что римские цифры удобны для сложения и вычитания, именно поэтому они так долго использовались в Европе при ведении бухгалтерии. Действительно, при сложении двух римских чисел просто выписываются рядом все символы из двух этих чисел, а затем применяется всего несколько правил: пять *I* образуют *V*, две *V* — *X*, пять *X* — *L* и т. д.

Сложно умножать и делить числа, записанные римскими цифрами. Многие другие ранние числовые системы (например, греческая) аналогично не подходят для сложных математических действий. Древние греки разработали превосходную геометрию, которая до сих пор практически без изменений преподается в школах, но так ли известна древнегреческая алгебра?

Цифры, которыми мы пользуемся сегодня, называются индо-арабскими. Они возникли в Индии, но были занесены в Европу арабскими математиками. Особенно прославился персидский математик Мухаммад ибн Муса аль-Хорезми (от имени которого происходит слово «*алгоритм*»), написавший около 825 года книгу по алгебре, где пользовался индийскими цифрами. Эта книга была переведена на латынь около 1120 года, оказала большое влияние на Европу и стимулировала переход с римских цифр на современные.

Индо-арабская система чисел отличалась от более ранних.

— Индо-арабская система называется *позиционной*, то есть любая цифра может обозначать в ней разное количество в зависимости от того, в какой части числа стоит. Положение цифры в числе не менее (и даже более) важно, чем значение самой цифры. И в 100, и в 1 000 000 всего по одной единице, но всем известно, что миллион гораздо больше сотни.
— Практически во всех ранних системах счисления было нечто, чего *нет* в индо-арабской системе, а именно: отдельный символ для обозначения десятки. В нашей системе счисления такой символ *отсутствует*.
— С другой стороны, практически во всех ранних числовых системах отсутствовало кое-что, имеющееся в индо-арабской системе и, по сути, более важное, чем символ десятки, — символ нуля.

Да, ноль. Скромный ноль, несомненно, одно из важнейших изобретений в истории чисел и математики. Он обеспечивает позиционную запись, поскольку позволяет отличить 25 от 205 и от 250. Ноль упрощает многие математические действия, неудобные в непозиционных системах, особенно умножение и деление.

Вся структура индо-арабских чисел проясняется, если обратить внимание, как мы их произносим. Например, 4825: «Четыре тысячи восемьсот двадцать пять». Это означает:

четыре тысячи,
восемь сотен,
два десятка
и еще пять.

Либо можно разложить это число на компоненты, например:

$$4825 = 4000 + 800 + 20 + 5.$$

Или еще мельче, вот так:

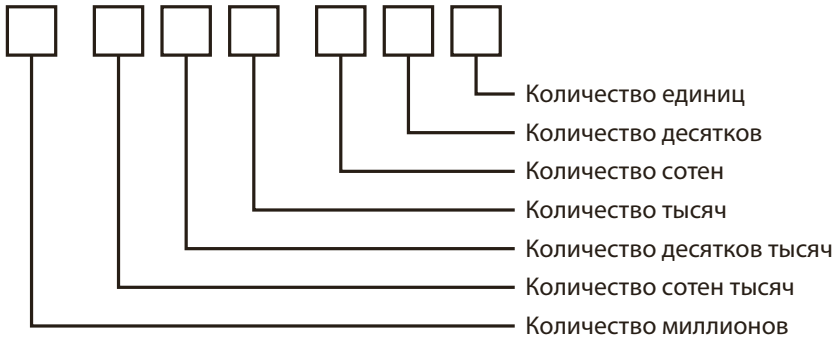
$$\begin{aligned} 4825 = {} & 4 \times 1000 + \\ & 8 \times 100 + \\ & 2 \times 10 + \\ & 5 \times 1. \end{aligned}$$

Или, воспользовавшись степенями десятки, записать следующее:

$$4825 = 4 \times 10^3 + 8 \times 10^2 + 2 \times 10^1 + 5 \times 10^0.$$

Напоминаю: любое число в степени 0 равно единице.

Каждая позиция в многозначном числе имеет определенное значение, как показано на следующей схеме. В семи окошках можно записать любое число от 0 до 9 999 999.



Каждая позиция соответствует степени десятки. Специального символа для десятки не требуется, поскольку 1 просто ставится в нужную позицию, а 0 используется в качестве символа-заполнителя.

Самое замечательное в данном случае в том, что дробные величины, обозначаемые цифрами после десятичной запятой, подчиняются той же закономерности. Число 42 705,684 равно:

$$4 \times 10\,000 + 2 \times 1000 + 7 \times 100 + 0 \times 10 + 5 \times 1 + 6 \div 10 + 8 \div 100 + 4 \div 1000.$$

Это число можно записать и без деления:

$$4 \times 10\,000 +$$
$$2 \times 1000 +$$
$$7 \times 100 +$$
$$0 \times 10 +$$
$$5 \times 1 +$$
$$6 \times 0{,}1 +$$
$$8 \times 0{,}01 +$$
$$4 \times 0{,}001.$$

Или при помощи степеней десятки:

$$4 \times 10^4 +$$
$$2 \times 10^3 +$$
$$7 \times 10^2 +$$
$$0 \times 10^1 +$$
$$5 \times 10^0 +$$
$$6 \times 10^{-1} +$$
$$8 \times 10^{-2} +$$
$$4 \times 10^{-3}.$$

Обратите внимание: сначала степень доходит до нуля, а затем получает отрицательные значения.

Известно, что 3 плюс 4 равно 7. Аналогично 30 плюс 40 равно 70, 300 плюс 400 равно 700 и 3000 плюс 4000 равно 7000. В этом и заключается красота индо-арабской системы. Складывая сколь угодно длинные десятеричные числа, мы фактически решаем эту задачу поэтапно. На каждом этапе мы всего лишь складываем однозначные числа. Именно поэтому кто-то давным-давно заставлял вас запоминать таблицу сложения.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13

+	0	1	2	3	4	5	6	7	8	9
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Найдите в верхнем ряду и в левом столбце два числа, которые хотите сложить. Следуйте от них по прямой к центру, пока линии не пересекутся, и получите сумму. Например, 4 плюс 6 равно 10.

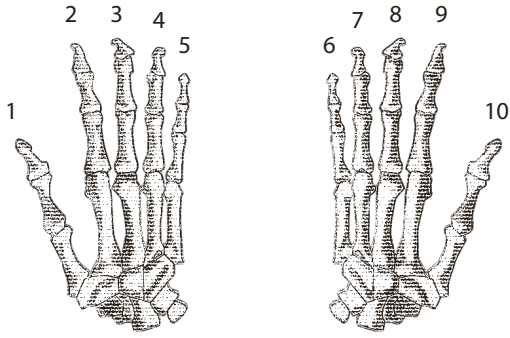
Аналогично, если требуется перемножить два десятеричных числа, выполняется более сложная процедура, которая тем не менее подразделяется на мелкие этапы, связанные с перемножением однозначных десятеричных чисел. Помните, в начальной школе мы должны были учить и таблицу умножения.

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

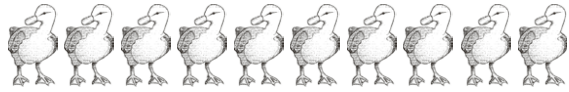
Главная прелесть позиционной нотации не в том, как хорошо она работает, а в том, как хорошо она применима в системах счисления, основанных не на десятке. Наша система счисления кому-то может показаться неудобной. Например, у большинства героев-мультяшек всего по четыре пальца на руке (или на лапе), поэтому им было бы сподручнее пользоваться системой с основанием 8. Довольно интересно следующее: бóльшая часть правил, известных нам по десятеричной системе, вполне применима и в восьмеричной.

Глава 8
Альтернативы десятке

Число 10 — исключительно важное для человека. У большинства из нас по десять пальцев на руках и на ногах, и мы, конечно, предпочитаем, чтобы и тех, и других было по десять. Поскольку на пальцах удобно считать, человек выстроил всю систему счисления на основании числа 10.



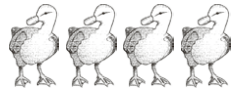
Как упоминалось в предыдущей главе, такая система называется «*система с основанием 10*», или «*десятеричная*». Она кажется нам столь естественной, что поначалу сложно даже найти альтернативу. Действительно, когда видим число 10, нас тянет представить, что оно означает, например, десять уток.

10= 

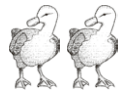
Единственная причина, по которой возникает такая ассоциация, в том, что уток столько же, сколько и пальцев у нас на руках. Если бы у человека было иное количество пальцев, то и считали бы мы по-другому, и число 10 означало бы нечто иное. Например, число 10 может указывать и на такое количество уток.

10= 

Или так.

10= 

Или даже так.

10= 

Как только мы поймем, в каком случае 10 означает двух уток, можно будет приступать к разговору о представлении чисел при работе с переключателями, проводами, лампочками и реле (далее — и с компьютерами).

Что, если бы у людей было всего по четыре пальца на руке, как у мультяшек? Вероятно, нам бы даже не пришло в голову разрабатывать десятеричную систему счисления. Напротив, мы бы считали нормальным, естественным, разумным, неизбежным, неопровержимым и бесспорно верным построить систему счисления с основанием 8. Она называлась бы не *десятеричной*, а *восьмеричной*, или *системой с основанием 8*.

Если бы наша система счисления была построена на основании 8, то вот этот символ нам бы не требовался:

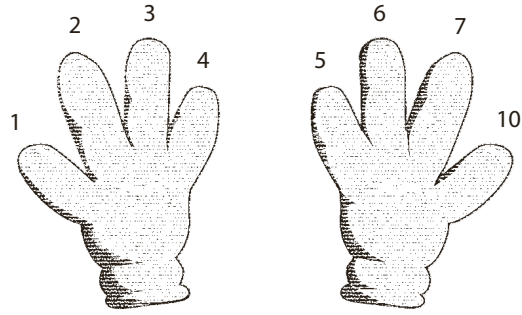
9.

Покажите этот символ мультяшке, и герой спросит: «Что это? Зачем это нужно?» Если задуматься, то и без этого символа можно обойтись:

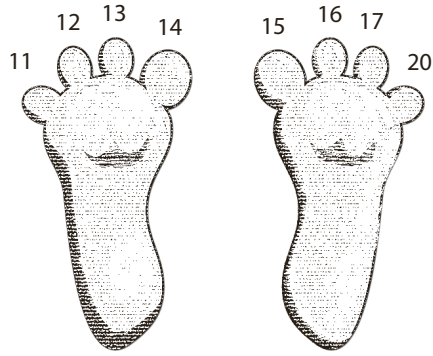
8.

В десятеричной системе счисления нет специального символа для десятки, соответственно в восьмеричной системе счисления его нет для восьмерки.

В десятеричной системе счисления мы считаем: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, а потом 10. В восьмеричной системе считаем: 0, 1, 2, 3, 4, 5, 6, 7, а потом что? Цифры кончились. Остается лишь 10, и это правильный ответ. В восьмеричной системе за 7 следует 10. Но в таком случае 10 соответствует не десяти пальцам, которые есть на двух руках у человека. В восьмеричной системе 10 — это количество пальцев у мультяшек.



Давайте считать дальше на четырехпалых ступнях.



Имея дело с иными системами счисления, кроме десятеричной, можно не путаться, если называть число 10 «*один-ноль*». Аналогично 13 будет «*один-три*», а 20 — «*два-ноль*». Чтобы *вообще* обойтись без путаницы, можно говорить «*два-ноль с основанием восемь*» или «*два-ноль восьмеричных*».

Даже когда у нас кончатся пальцы на руках и ногах, можно и далее считать в восьмеричной системе. В принципе, процесс не отличается от счета в десятеричной, просто мы пропускаем все числа, в которых есть 8 или 9. Естественно, конкретные числа обозначают уже другие величины.

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22,
23, 24, 25, 26, 27, 30, 31, 32, 33, 34, 35, 36, 37, 40, 41, 42, 43,
44, 45, 46, 47, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 64,
65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77, 100…

Последнее число называется «*один-ноль-ноль*». Это общее количество пальцев мультяшки, умноженное само на себя.

При записи десятеричных и восьмеричных чисел можно избежать путаницы, записывая все числа с нижними индексами, обозначающими принадлежность

к той или иной системе счисления. Нижний индекс ДЕСЯТЬ означает «основание десять», то есть десятеричную систему, а нижний индекс ВОСЕМЬ — «основание восемь», или восьмеричную систему.

Итак, Белоснежка повстречала $7_{\text{десять}}$, или $7_{\text{восемь}}$, гномов.
У мультяшек по $8_{\text{десять}}$, или $10_{\text{восемь}}$, пальцев на руке.
Бетховен написал $9_{\text{десять}}$, или $11_{\text{восемь}}$, симфоний.
У человека $10_{\text{десять}}$, или $12_{\text{восемь}}$, пальцев на руках.
В году $12_{\text{десять}}$, или $14_{\text{восемь}}$, месяцев.
В двух неделях $14_{\text{десять}}$, или $16_{\text{восемь}}$, дней.
Паспорт выдают в $16_{\text{десять}}$, или $20_{\text{восемь}}$, лет.
В сутках $24_{\text{десять}}$, или $30_{\text{восемь}}$, часов.
В латинице $26_{\text{десять}}$, или $32_{\text{восемь}}$, букв.
В английской кварте $907_{\text{десять}}$, или 1134, граммов.
В покерной колоде $52_{\text{десять}}$, или $64_{\text{восемь}}$, карт.
Самый известный адрес по Сансет-Стрип — $77_{\text{десять}}$, или $115_{\text{восемь}}$.
Длина поля для американского футбола — $91_{\text{десять}}$, или $131_{\text{восемь}}$, метров.
На старте женского одиночного зачета в Уимблдонском турнире — $128_{\text{десять}}$, или $200_{\text{восемь}}$, участниц.
Площадь Мемфиса равна $640_{\text{десять}}$, или $1000_{\text{восемь}}$, квадратных километров.

Обратите внимание: в этом списке есть несколько *круглых* восьмеричных чисел. Круглым называется число, оканчивающееся на один или несколько нулей. Если десятеричное число оканчивается двумя нулями, значит, оно кратно $100_{\text{десять}}$, а $100_{\text{десять}}$ — это $10_{\text{десять}}$, умноженное на $10_{\text{десять}}$. В восьмеричной системе два нуля в конце числа означают, что число кратно $100_{\text{восемь}}$, то есть $10_{\text{восемь}}$ умножить на $10_{\text{восемь}}$ (или $8_{\text{десять}}$ умножить на $8_{\text{десять}}$, что равно $64_{\text{десять}}$).

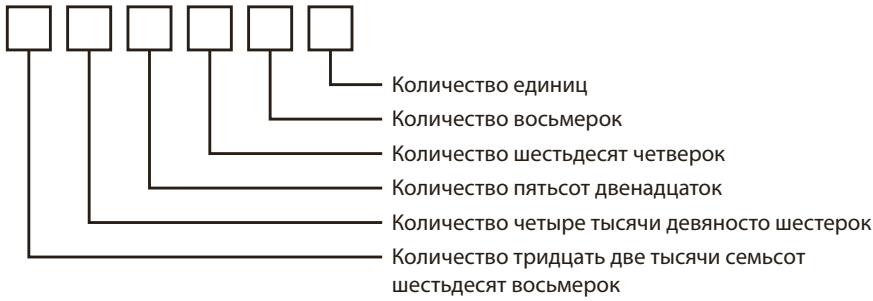
Возможно, вы также заметили, что такие круглые восьмеричные числа, как $100_{\text{восемь}}$, $200_{\text{восемь}}$ и $400_{\text{восемь}}$, в десятеричной системе соответствуют $64_{\text{десять}}$, $128_{\text{десять}}$ и $256_{\text{десять}}$, и все эти десятеричные числа — степени двойки. Это логично. Например, число $400_{\text{восемь}}$, равно $4_{\text{восемь}}$ умножить на $10_{\text{восемь}}$ и умножить на $10_{\text{восемь}}$, и все это — степени двойки. Всякий раз при умножении степени двойки на степень двойки мы получаем еще одну степень двойки.

В следующей таблице даны некоторые степени двойки в десятеричном и восьмеричном представлении.

Степень двойки	Десятеричная система	Восьмеричная система
2^0	1	1
2^1	2	2
2^2	4	4
2^3	8	10
2^4	16	20
2^5	32	40
2^6	64	100
2^7	128	200
2^8	256	400
2^9	512	1000
2^{10}	1024	2000
2^{11}	2048	4000
2^{12}	4096	10000

Круглые числа из правого столбца подсказывают, что системы счисления, отличающиеся от десятеричной, удобны для работы с двоичными кодами.

Структурно восьмеричная система аналогична десятеричной. Отличия лишь в деталях. Например, каждая позиция в восьмеричном числе — это цифра, умноженная на степень восьмерки.



Следовательно, восьмеричное число $3725_{\text{ВОСЕМЬ}}$ можно разбить:

$$3725_{\text{ВОСЕМЬ}} = 3000_{\text{ВОСЕМЬ}} + 700_{\text{ВОСЕМЬ}} + 20_{\text{ВОСЕМЬ}} + 5_{\text{ВОСЕМЬ}}.$$

Эту последовательность можно переписать несколько иначе. Например, при помощи степеней восьмерки в их десятеричном представлении:

$$\begin{aligned} 3725_{\text{ВОСЕМЬ}} = {} & 3 \times 512_{\text{ДЕСЯТЬ}} + \\ & 7 \times 64_{\text{ДЕСЯТЬ}} + \\ & 2 \times 8_{\text{ДЕСЯТЬ}} + \\ & 5 \times 1. \end{aligned}$$

То же самое, записанное при помощи степеней восьмерки в восьмеричном представлении:

$$3725_{\text{ВОСЕМЬ}} = 3 \times 1000_{\text{ВОСЕМЬ}} + 7 \times 100_{\text{ВОСЕМЬ}} + 2 \times 10_{\text{ВОСЕМЬ}} + 5 \times 1.$$

А можно сделать вот так:

$$3725_{\text{ВОСЕМЬ}} = 3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 5 \times 8^0.$$

Если выполнить эти расчеты в десятеричной системе, получится $2005_{\text{ДЕСЯТЬ}}$. Таким образом восьмеричные числа преобразуются в десятеричные.

Восьмеричные числа складываются и перемножаются в точности как десятеричные. Разница в том, что таблицы умножения и сложения для восьмеричных чисел строятся иначе. Вот таблица сложения восьмеричных чисел.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

Например, $5_{\text{ВОСЕМЬ}} + 7_{\text{ВОСЕМЬ}} = 14_{\text{ВОСЕМЬ}}$, то есть восьмеричные числа можно складывать в столбик.

$$\begin{array}{r} 135 \\ +\ 643 \\ \hline 1000 \end{array}$$

Начинаем справа: 5 плюс 3 равно 10, 0 пишем, 1 в уме; 1 плюс 3 плюс 4 равно 10, 0 пишем, 1 в уме; 1 плюс 1 плюс 6 равно 10.

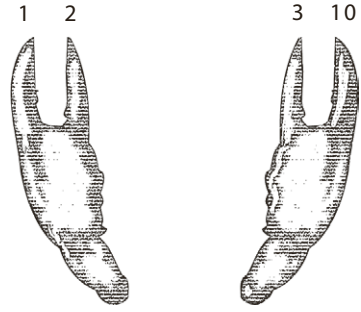
Аналогично дважды два и в восьмеричной системе равно четырем. Но трижды три не равно девяти. А как? Трижды три равно $11_{\text{восемь}}$, это столько же, сколько и $9_{\text{десять}}$. Далее полностью приведена восьмеричная таблица умножения.

×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Здесь у нас 4 × 6 равно $30_{\text{восемь}}$, но $30_{\text{восемь}}$ равно $24_{\text{десять}}$, то есть 4 × 6 в десятеричной системе.

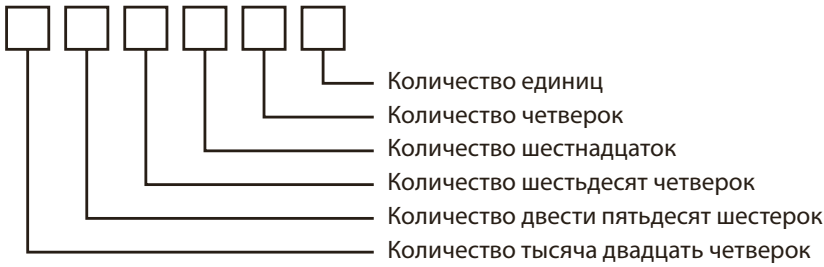
Восьмеричная система счисления столь же полноценна, как и десятеричная.

Мы разработали систему счисления для мультяшек. Теперь давайте создадим такую же систему для омаров. У омаров нет пальцев, но на кончиках передних лап у них клешни. Омарам подойдет четверичная система счисления с основанием четыре.



Вот как считают в четверичной системе: 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110 и т. д.

Не буду подробно останавливаться на четверичной системе, поскольку мы приближаемся к более важному вопросу. Как видите, здесь каждая позиция в числе соответствует степени четверки.



В четверичной системе счисления число 31 232 можно записать следующим образом:

$$31\,232_{\text{четыре}} = 3 \times 256_{\text{десять}} + 1 \times 64_{\text{десять}} + 2 \times 16_{\text{десять}} + 3 \times 4_{\text{десять}} + 2 \times 1_{\text{десять}}.$$

Что равнозначно записи:

$$31\,232_{\text{четыре}} = 3 \times 10\,000_{\text{четыре}} + 1 \times 1000_{\text{четыре}} + 2 \times 100_{\text{четыре}} + 3 \times 10_{\text{четыре}} + 2 \times 1_{\text{четыре}}.$$

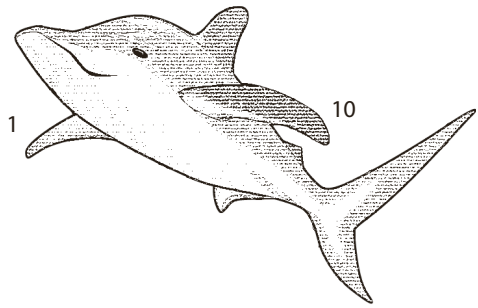
А это то же самое, что и:

$$31\,232_{\text{четыре}} = 3 \times 4^4 + 1 \times 4^3 + 2 \times 4^2 + 3 \times 4^1 + 2 \times 4^0.$$

Если мы выполним вычисления в десятичной системе счисления, то обнаружим, что $31\,232_{\text{четыре}}$ — это $878_{\text{десять}}$.

Теперь мы сделаем еще один прыжок, на этот раз окончательный. Представьте, что мы дельфины и можем использовать для подсчета два плавника. В данном случае мы имеем дело с системой счисления с основанием 2, или *двоичной*, или, иначе, *бинарной* (от лат. binary — «двойной», «состоящий из двух частей»). Понятно, что у нас будет только две цифры: 0 и 1.

С нулем и единицей мало что можно сделать, и, чтобы привыкнуть к двоичным числам, требуется практика. Проблема в том, что сразу заканчиваются цифры. Например, на следующем рисунке показано, как дельфин считает на плавниках.



Да, в двоичной системе счисления за 1 следует 10. Это странно, однако это не должно удивлять. Независимо от того, какую систему счисления мы используем, всякий раз, когда у нас заканчиваются отдельные цифры, первое двузначное число всегда 10. В двоичной системе счисления мы считаем:

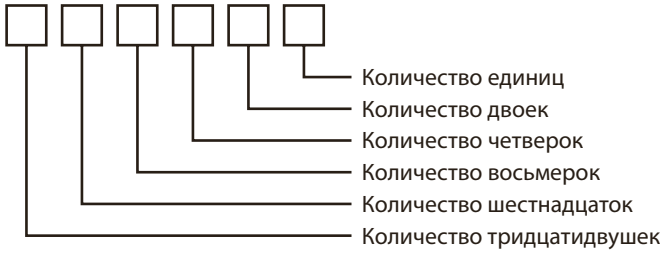
0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100,
1101, 1110, 1111, 10000, 10001…

Эти числа могут показаться большими, но на самом деле это не так. Скорее, двоичные числа очень быстро становятся *длинными*, а не большими.

Количество голов у людей — $1_{\text{десять}}$, или $1_{\text{два}}$.
Количество плавников у дельфинов — $2_{\text{десять}}$, или $10_{\text{два}}$.
Количество чайных ложек в столовой ложке — $3_{\text{десять}}$, или $11_{\text{два}}$.
Количество сторон у квадрата — $4_{\text{десять}}$, или $100_{\text{два}}$.
Количество пальцев на одной человеческой руке — $5_{\text{десять}}$, или $101_{\text{два}}$.
Количество конечностей у насекомых — $6_{\text{десять}}$, или $110_{\text{два}}$.
Количество дней в неделе — $7_{\text{десять}}$, или $111_{\text{два}}$.
Количество музыкантов в октете — $8_{\text{десять}}$, или $1000_{\text{два}}$.

Количество планет в Солнечной системе, включая Плутон, — $9_{\text{десять}}$, или $1001_{\text{два}}$.
Количество центнеров в тонне — $10_{\text{десять}}$, или $1010_{\text{два}}$.

В двоичном числе, состоящем из большого количества цифр, позиции знаков соответствуют степени двойки.



Таким образом, каждый раз, когда встречаем двоичное число, состоящее из единицы и следующих за ней нулей, мы понимаем, что это число соответствует какой-либо из степеней двойки. Эта степень равна количеству нулей в этом двоичном числе. Вот наша расширенная таблица степеней двойки, демонстрирующая такое правило.

Степень двойки	**Десятичная система**	**Восьмеричная система**	**Четверичная система**	**Двоичная система**
2^0	1	1	1	1
2^1	2	2	2	10
2^2	4	4	10	100
2^3	8	10	20	1000
2^4	16	20	100	10000
2^5	32	40	200	100000
2^6	64	100	1000	1000000
2^7	128	200	2000	10000000
2^8	256	400	10000	100000000
2^9	512	1000	20000	1000000000
2^{10}	1024	2000	100000	10000000000
2^{11}	2048	4000	200000	100000000000
2^{12}	4096	10000	1000000	1000000000000

Допустим, у нас есть двоичное число 101101011010. Его можно записать так:

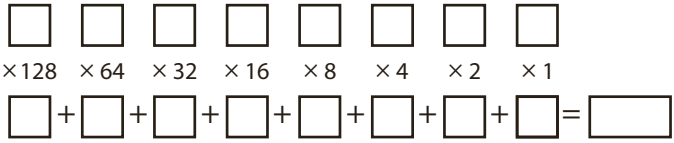
$$
\begin{aligned}
101101011010_{\text{ДВА}} = {} & 1 \times 2048_{\text{ДЕСЯТЬ}} + \\
& 0 \times 1024_{\text{ДЕСЯТЬ}} + \\
& 1 \times 512_{\text{ДЕСЯТЬ}} + \\
& 1 \times 256_{\text{ДЕСЯТЬ}} + \\
& 0 \times 128_{\text{ДЕСЯТЬ}} + \\
& 1 \times 64_{\text{ДЕСЯТЬ}} + \\
& 0 \times 32_{\text{ДЕСЯТЬ}} + \\
& 1 \times 16_{\text{ДЕСЯТЬ}} + \\
& 1 \times 8_{\text{ДЕСЯТЬ}} + \\
& 0 \times 4_{\text{ДЕСЯТЬ}} + \\
& 1 \times 2_{\text{ДЕСЯТЬ}} + \\
& 0 \times 1_{\text{ДЕСЯТЬ}}.
\end{aligned}
$$

Или:

$$
\begin{aligned}
101101011010_{\text{ДВА}} = {} & 1 \times 2^{11} + \\
& 0 \times 2^{10} + \\
& 1 \times 2^{9} + \\
& 1 \times 2^{8} + \\
& 0 \times 2^{7} + \\
& 1 \times 2^{6} + \\
& 0 \times 2^{5} + \\
& 1 \times 2^{4} + \\
& 1 \times 2^{3} + \\
& 0 \times 2^{2} + \\
& 1 \times 2^{1} + \\
& 0 \times 2^{0}.
\end{aligned}
$$

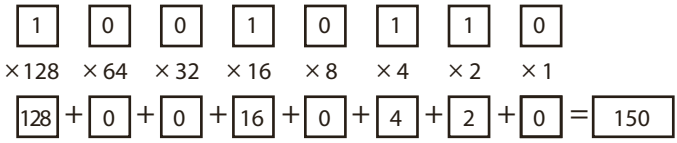
Если просто сложить все слагаемые в десятичной системе, получим 2048 + 512 + 256 + 64 + 16 + 8 + 2, что составляет $2906_{\text{ДЕСЯТЬ}}$.

Для более легкого преобразования двоичных чисел в десятичные можно использовать следующую схему.

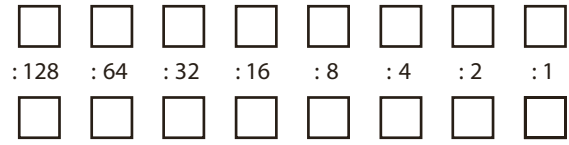


Эта схема позволяет конвертировать числа, содержащие до восьми двоичных разрядов; ее можно легко расширить. Введите до восьми цифр в восемь

верхних полей, по одной цифре в каждый квадрат. Выполните восемь операций умножения и введите их результаты в восемь нижних полей. Сложите числа в этих восьми полях для получения окончательного результата. Этот пример демонстрирует процесс нахождения десятичного эквивалента двоичного числа 10010110.

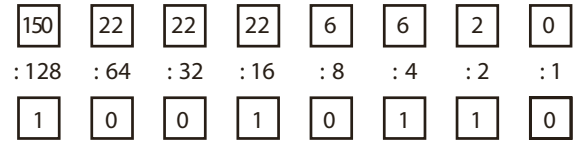


Преобразовать десятичные числа от 0 до 255 в двоичные не так просто, однако вы можете использовать следующую схему.



Процесс преобразования сложнее, чем кажется, поэтому внимательно следуйте указаниям. Поместите десятичное число (меньшее или равное 255) в верхний левый квадрат. Разделите это число (делимое) на первый делитель (128), как показано на схеме. Поместите целую часть в нижнее поле (левый нижний квадрат), а остаток от деления — в поле справа (второй квадрат в верхнем ряду). Этот первый остаток является делимым, которое будет участвовать в следующей операции деления, где в качестве делителя используется число 64.

Помните, что каждая целая часть будет равна либо 0, либо 1. Если делимое меньше делителя, то целая часть от деления будет равна 0, а остаток — самому делимому. Если делимое больше или равно делителю, то целая часть от деления будет равна 1, а остаток — разности между делимым и делителем. Вот как преобразуется число 150.



Если вам нужно сложить или перемножить два двоичных числа, вероятно, будет легче выполнить вычисления в двоичной системе, не преобразуя числа в десятичные. Это должно понравиться. Представьте, как быстро вы могли бы освоить сложение, если бы потребовалось запомнить только это.

+	**0**	**1**
0	0	1
1	1	10

Давайте с помощью этой таблицы сложим два двоичных числа.

$$\begin{array}{r} 1100101 \\ +\ 0110110 \\ \hline 10011011 \end{array}$$

Начиная с правого столбца: 1 плюс 0 равно 1. Второй столбец справа: 0 плюс 1 равно 1. Третий столбец: 1 плюс 1 равно 0, 1 в уме. Четвертый столбец: 1 (перенесенное значение) плюс 0 плюс 0 равно 1. Пятый столбец: 0 плюс 1 равно 1. Шестой столбец: 1 плюс 1 равно 0, 1 в уме. Седьмой столбец: 1 (перенесенное значение) плюс 1 плюс 0 = 10.

Таблица умножения даже проще, чем таблица сложения, поскольку ее можно составить, используя два базовых правила умножения: умножая на 0, получаем 0, умножение на 1 не влияет на исходное число.

×	**0**	**1**
0	0	0
1	0	1

Вот процесс умножения числа $13_{\text{десять}}$ на число $11_{\text{десять}}$ в двоичной системе счисления.

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ 1101 \\ \hline 10001111 \end{array}$$

Результат — $143_{\text{десять}}$.

Люди, работающие с двоичными числами, часто предваряют их нулями, то есть пишут нули слева от первой 1, например 0011 вместо 11. Это совершенно не влияет на значение, а служит исключительно для красоты. В следующей таблице перечислены первые шестнадцать двоичных чисел и их десятичные эквиваленты.

Двоичное число	**Десятичное число**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Давайте рассмотрим список двоичных чисел. Обратите внимание на каждый из четырех вертикальных столбцов, состоящих из нулей и единиц, и заметьте, как эти цифры чередуются в столбцах сверху вниз:

— в крайнем правом столбце — 0 и 1;
— во втором столбце справа — два 0 и две 1;
— в следующем столбце — четыре 0 и четыре 1;
— в крайнем левом столбце — восемь 0 и восемь 1.

В этом есть порядок, не так ли? Действительно, вы можете легко написать следующие шестнадцать двоичных чисел, просто повторив первые шестнадцать и добавив 1 в начале.

Двоичное число	Десятичное число
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

Вот еще один способ смотреть на это: при выполнении подсчета в двоичном формате крайняя цифра справа (также называемая младшим разрядом) поочередно принимает значения 0 и 1. Каждый раз, когда она изменяется с 1 на 0, вторая цифра справа, следующая за младшим разрядом, также изменяется либо с 0 на 1, либо с 1 на 0. Так что каждый раз, когда двоичная цифра изменяется с 1 на 0, следующая за ней цифра также меняется либо с 0 на 1, либо с 1 на 0.

При записи больших десятичных чисел мы используем запятые через каждые три знака для облегчения их восприятия*. Например, если вы увидите число 12000000, вероятно, придется подсчитать количество цифр, однако, увидев число 12,000,000, вы сразу поймете, что оно означает 12 миллионов.

Двоичные числа очень быстро могут стать весьма длинными. Например, 12 миллионов в двоичной системе счисления записывается так: 101101110001101100000000. Чтобы такое число было легче воспринимать, каждые четыре двоичных разряда обычно разделяются пробелами (1011 0111 0001 1011 0000 0000). Далее в этой книге мы рассмотрим более сжатый способ записи двоичных чисел.

Сведя систему счисления к двоичным цифрам 0 и 1, мы достигли предела. Далее упрощать некуда. Более того, двоичная система соединяет арифметику

* Запятая для разделения разрядов числа используется преимущественно в англоязычной нотации; дробная часть числа в таком случае отделяется точкой. В России принято разделять разряды пробелами, а дробную часть отделять запятой. *Прим. науч. ред.*

с электричеством. В предыдущих главах мы рассматривали переключатели, провода, лампочки и реле, и любой из этих объектов может отображать двоичные цифры 0 и 1.

Провод может представлять собой двоичную цифру. Если по нему идет ток, то двоичная цифра равна 1, если нет — 0.

Переключатель может представлять собой двоичную цифру. Если переключатель включен, или замкнут, то двоичная цифра равна 1, если переключатель выключен, или разомкнут, то двоичная цифра — 0.

Лампочка может представлять собой двоичную цифру. Если лампочка горит, то двоичная цифра равна 1, если нет — 0.

Телеграфное реле может представлять собой двоичную цифру. Если реле замкнуто, то двоичная цифра равна 1, если разомкнуто — 0.

Двоичные цифры имеют непосредственное отношение к компьютерам.

Примерно в 1948 году американский математик Джон Тьюки (род. 1915)* осознал, что в будущем словосочетание «двоичная цифра» (binary digit), вероятно, приобретет гораздо большее значение — по мере распространения компьютеров. Он решил создать новое, более короткое слово, чтобы заменить эти громоздкие пять слогов, и рассматривал такие варианты, как bigit и binit, но остановился на коротком, простом, элегантном и просто замечательном слове bit («бит»).

---

* Вскоре после выхода в свет первого издания этой книги, в том же 2000 году, Джон Тьюки скончался.

Глава 9
За битом бит

Когда в 1973 году Тони Орландо в своей песне попросил, чтобы любимая «повязала желтую ленточку вокруг старого дуба», он не сопроводил свою просьбу ни подробными объяснениями, ни долгими рассуждениями. Никаких «если», «и», «но». Несмотря на сложные чувства и эмоции, сопровождавшие ситуацию, что разворачивалась в реальной жизни и легла в основу песни, мужчине хотелось получить простой ответ: «да» или «нет». Он знал, что, если на старом дубе появится желтая ленточка, это будет означать: «Да, хотя ты и наворотил дел и провел три года в тюрьме, я все равно хочу, чтобы ты вернулся и мы жили под одной крышей». А отсутствие желтой ленточки скажет: «Даже не думай здесь останавливаться».

Здесь есть две четкие взаимоисключающие альтернативы. Тони Орландо не пел: «Повяжи половину желтой ленточки, если тебе нужно время на размышление» или «Повяжи голубую ленточку, если больше не любишь меня, но по-прежнему хочешь остаться друзьями». Нет, он все сформулировал очень просто.

Не менее информативно, нежели отсутствие или наличие желтой ленточки (разве что не столь поэтично), сработал бы дорожный знак, поставленный во дворе, например «Путь открыт» или «Въезд запрещен». Или табличка на двери «Закрыто» или «Открыто». Или фонарик на окне — включенный или выключенный.

Если вам нужно просто сказать «да» или «нет», то способов хватает. Для этого не надо произносить ни одной фразы, слова, даже буквы. Необходим всего один *бит*, то есть 0 или 1.

Как мы узнали в предыдущих главах, десятеричная система, которой мы пользуемся при подсчете предметов, на самом деле ничем не примечательна. Ясно, что мы пользуемся системой с основанием 10, так как у нас 10 пальцев на руках. Мы могли бы с тем же успехом использовать систему с основанием 8 (будь мы мультяшками), 4 (будь мы омарами) или даже 2 (если бы мы были дельфинами).

Однако двоичная система счисления *все-таки* особенная. Дело в том, что это *простейшая* возможная система счисления. В двоичной системе всего две

цифры: 0 и 1. Если нам нужно что-то проще двоичной системы, придется избавиться от 1, и останется только 0. Имея всего лишь 0, ничего не сделаешь.

Слово «бит» — сокращение от английского выражения binary digit («двоичная цифра»). Пожалуй, это одно из симпатичнейших слов в компьютерной терминологии. В английском языке у слова bit есть и общеупотребительное значение — «кусочек, небольшая часть», и это значение нам отлично подходит, поскольку один бит — двоичная цифра, мельчайший фрагмент информации.

Иногда, если изобретается новое слово, ему присваивается новое значение. В данном случае все именно так. Слово «бит» — это не только двоичные цифры, при помощи которых удобно считать дельфинам. В компьютерную эпоху это слово приобрело значение «*мельчайший первичный фрагмент информации*».

Да, смелое утверждение. Естественно, информация передается не только при помощи битов. Буквы, слова, азбука Морзе и шрифт Брайля, а также десятеричные цифры тоже переносят информацию. Суть бита заключается в том, что он передает очень *мало* информации. Один бит — это минимально возможное количество информации. Меньше бита — отсутствие информации.

Поскольку бит выражает минимальный возможный объем информации, более сложную информацию можно передать некоторым количеством битов. (Говоря, что бит передает мало информации, я совершенно не имею в виду, что эта информация граничит с бессмыслицей. Желтая ленточка *очень* важна для тех, кто в ней заинтересован.)

«Запомните, дети, — слышал весь мир, // Как в полночь глухую скакал Поль Ревир…» — писал Генри Лонгфелло. Возможно, он исторически недостоверно описал поступок Поля Ревира, предупредившего американских колонистов о вторжении британцев, однако эти стихи замечательно демонстрируют, как можно передать важную информацию при помощи битов.

Он другу сказал: «Я сигнала жду.
Коль ночью из города наступать
Начнут британцы, ты дай мне знать,
На Северной церкви зажги звезду, —
Одну, если сушей, а морем — две».*

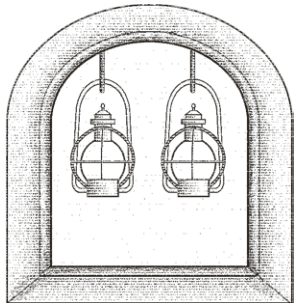
Итак, Ревир дал другу два фонаря. Если британцы вторгнутся по суше, друг зажжет на колокольне один, если высадятся с моря — то зажжет оба.

Правда, Лонгфелло не дает явного указания на все случаи жизни. Он умалчивает о *третьей* возможности, означающей, что британцы пока не вторгаются.

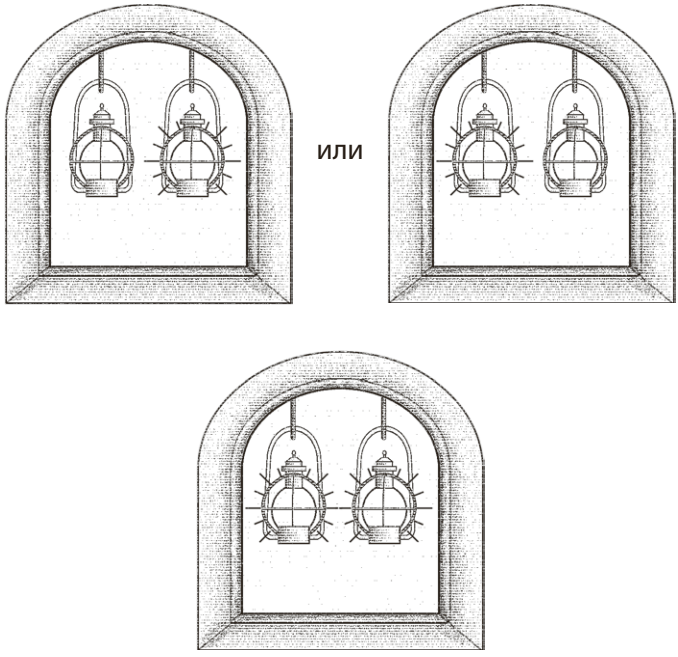
* Здесь и далее стихотворение приводится в переводе М. А. Зенкевича.

Лонгфелло подразумевает, что тогда ни одного фонаря на колокольне гореть не будет.

Допустим, два фонаря висят на колокольне всегда. Как правило, они не горят.



Значит, британцы пока не наступают. Если зажжен один фонарь, значит, британцы наступают по суше, если горят оба, — высаживаются с моря.



Каждый фонарь — это один бит. Зажженный фонарь равен единичному биту, незажженный — нулевому биту. Тони Орландо показал, что для описания любой из двух возможностей достаточно всего одного бита. Если бы Поль Ревир должен был оповестить людей лишь о том, вторгаются британцы или нет (а не откуда именно), то хватило бы и одного фонаря. Фонарь горел бы в случае нападения и был бы потушен, если бы предстоял еще один спокойный вечер.

Чтобы описать одну из трех возможностей, требуется два фонаря. При наличии второго фонаря и двух битов уже можно описать четыре возможности:

00 = британцы пока не наступают;
01 = британцы наступают с суши;
10 = британцы наступают с суши;
11 = британцы наступают с моря.

На самом деле Поль Ревир, ограничившись всего тремя возможностями, поступил весьма хитроумно. В телекоммуникационной терминологии можно сказать, что он задействовал *избыточность*, позволяющую бороться с *шумом*. Термин «шум» в теории телекоммуникаций означает любые помехи, осложняющие передачу информации. Типичный пример шума — помехи на телефонной линии. Тем не менее, даже несмотря на помехи, общение по телефону обычно проходит успешно, поскольку устная речь крайне избыточна. Не требуется четко слышать каждый слог в каждом слове, чтобы понять, что сказали.

В случае с фонарями на колокольне к «шуму» можно отнести такие факторы, как темнота и расстояние от Поля Ревира до колокольни, которые могут помешать отличить первый фонарь от второго. Вот важнейший отрывок из стихов Лонгфелло:

Вдруг видит — звездой огонек замигал,
То дан с колокольни желанный сигнал!
В седло он вскочил, сжал повод в ладонь.
Под ним захрапел в нетерпенье конь.
Второй сигнал! Он коня погнал!

Скорее всего, Поль Ревир был не в состоянии определить, какой фонарь зажегся первым.

Из этого следует важный вывод: *информация — выбор из двух или более возможностей*. Например, когда мы говорим с кем-либо, любое произносимое слово выбирается из словаря. Если бы мы пронумеровали все слова из словаря от 1 до 351 482, то могли бы столь же четко вести беседу, называя номера, а не слова (естественно, обоим собеседникам понадобились бы словари, где все слова пронумерованы одинаково, а также терпение).

Верно и обратное: *любую информацию можно свести к выбору между двумя или более возможностями*, а сами возможности выразить при помощи битов. Излишне говорить, что существует множество таких вариантов человеческой коммуникации, где выбор между конкретными возможностями

не предоставляется, и такие формы коммуникации также жизненно необходимы. Вот почему люди не флиртуют с компьютерами (по крайней мере, хочется на это надеяться). Если некоторую информацию невозможно выразить при помощи слов, картинок или звуков, значит, ее нельзя закодировать при помощи битов. Да и не возникает такого желания.

Жесты «палец вверх» или «палец вниз» содержат по одному биту информации. Еще есть жесты «два пальца вверх» и «два пальца вниз». Два последних жеста нравились ныне покойным кинокритикам Роджеру Эберту и Джину Сискелу, которые таким образом выносили окончательные вердикты по новейшим фильмам. Итак, в данном случае мы следим только за их большими пальцами. Получается четыре возможности, которые можно представить в виде двух битов:

00 = обоим не понравилось;
01 = Сискелу не понравилось, Эберту понравилось;
10 = Сискелу понравилось, Эберту не понравилось;
11 = обоим понравилось.

Первый бит относится к Сискелу; таким образом, 0 означает, что кино не понравилось Сискелу, а 1 — что кино понравилось Сискелу. Аналогично второй бит описывает вкусы Эберта. Итак, если друг спросит, что решили Сискел и Эберт о фильме Impolite Encounter, вы, вместо того чтобы ответить: «С точки зрения Сискела — большой палец вверх, с точки зрения Эберта — большой палец вниз» или «Сискелу понравилось; Эберту — нет», можете просто сказать: «Один ноль». Если друг знает, какой бит относится к Сискелу, а какой — к Эберту, а 1 означает «палец вверх», 0 — «палец вниз», то ваш ответ будет понятен. Просто вы должны знать код.

Можно сразу условиться, что бит 1 означает «палец вниз», а 0 — «палец вверх». Это может показаться нелогичным. Естественно, мы привыкли считать, что 1 означает нечто «утвердительное», а 0 — наоборот. На самом деле такое соответствие произвольное. Требуется всего лишь, чтобы все, кто пользуется кодом, знали значения бита 0 и 1.

Значение конкретного бита или совокупности битов всегда понимается в контексте. Вероятно, смысл желтой ленточки, повязанной на конкретном дубе, понятен лишь тому, кто ее туда повесил, и тому, для кого предназначен этот знак. Достаточно изменить цвет, дерево, дату — и значение исчезнет, останется просто тряпочка. Чтобы извлечь какую-либо полезную информацию из жестикуляции Сискела и Эберта, нужно как минимум понимать, о каком фильме идет речь.

Если вы вели список фильмов, отрецензированных Сискелом и Эбертом, и фиксировали их голоса, можно добавить в систему еще один бит, который будет выражать ваше мнение. В таком случае количество возможностей доходит до восьми:

000 = Сискелу не понравилось; Эберту не понравилось; мне не понравилось;
001 = Сискелу не понравилось; Эберту не понравилось; мне понравилось;
010 = Сискелу не понравилось; Эберту понравилось; мне не понравилось;
011 = Сискелу не понравилось; Эберту понравилось; мне понравилось;
100 = Сискелу понравилось; Эберту не понравилось; мне не понравилось;
101 = Сискелу понравилось; Эберту не понравилось; мне понравилось;
110 = Сискелу понравилось; Эберту понравилось; мне не понравилось;
111 = Сискелу понравилось; Эберту понравилось; мне понравилось.

Один из плюсов при представлении этой информации в виде битов таков: мы уверены, что учли все возможности, мы знаем, что существует восемь и только восемь возможностей — ни больше, ни меньше. Имея три бита, можно сосчитать лишь от нуля до семи. Трехзначных двоичных чисел больше нет.

Итак, при описании таких битов Сискела и Эберта вас, возможно, уже занимает следующий непростой вопрос: «А что насчет сборника рецензий Leonard Maltin's Movie & Video Guide?» Ведь Леонард Малтин оценивает фильмы не такими «пальцевыми» жестами, а более традиционно — при помощи системы звезд.

Чтобы определить, сколько битов Малтина требуется, сперва необходимо разобраться в его системе. Малтин оценивает фильм в некоторое количество звезд (от одной до четырех), причем звезда может делиться и пополам. (Кстати, одну звезду Мартин никогда не присваивает; вместо этого такой фильм получает рейтинг КРАХ.) Вот семь возможностей, и это означает, что для представления любой оценки достаточно трех бит:

000 = КРАХ;
001 = ★★;
010 = ★★;
011 = ★★★;
100 = ★★★;
101 = ★★★★;
110 = ★★★★.

«Что же насчет 111?» — могли бы спросить вы. Да, этот код ничего не значит. Он не определен. Если бы мы использовали двоичный код 111 в рейтингах Малтина, это бы свидетельствовало об ошибке. (Вероятно, ошибся компьютер, ведь люди никогда не ошибаются!)

Напомню, что, когда мы представляли оценки Сискела и Эберта при помощи двух битов, левый бит относился к Сискелу, правый — к Эберту. Есть ли в данном случае какое-либо значение у отдельных битов? Да, в некотором роде. Если взять числовое значение такого битового кода, прибавить к нему 2, а затем разделить результат на 2, то получится количество звезд. Это возможно лишь потому, что мы определяли коды продуманно и непротиворечиво. Мы вполне могли определить коды и так:

000 = ★★★;
001 = ★½;
010 = ★★½;
011 = ★★★★;
101 = ★★★½;
110 = ★★;
111 = КРАХ.

Этот код столь же адекватен, как и предыдущий, если всем понятно его значение.

Если бы Малтину попался фильм, не заслуживающий даже единственной звезды, он мог бы присвоить ему рейтинг в половину звезды. Определенно, кодов на это ему бы хватило. В таком случае коды следовало бы перераспределить так:

000 = БОЛЬШОЙ КРАХ;
001 = КРАХ;
010 = ★½;
011 = ★★;
100 = ★★½;
101 = ★★★;
110 = ★★★½;
111 = ★★★★.

Однако если бы затем нашелся такой провальный фильм, который даже половины звезды не заслуживает (МЕГАКРАХ?), то понадобился бы еще один бит: 3-битных кодов больше не осталось.

В журнале Entertainment Weekly оцениваются не только фильмы, но и телешоу, CD, книги, сайты. Оценки варьируются от A+ до F (правда, есть ощущение, что такой чести удостаиваются лишь фильмы Поли Шора). Если подсчитать все возможные оценки, их наберется тринадцать. Для представления этой системы понадобится четыре бита:

0000 = F;
0001 = D–;
0010 = D;
0011 = D+;
0100 = C–;
0101 = C;
0110 = C+;
0111 = B–;
1000 = B;
1001 = B+;
1010 = A–;
1011 = A;
1100 = A+.

У нас осталось три неиспользованных кода: 1101, 1110 и 1111, а всего их шестнадцать.

Рассуждая о битах, мы часто говорим о конкретном *числе* битов. Чем больше битов у нас в распоряжении, тем больше возможностей удается с их помощью описать.

Естественно, с десятеричными числами складывается точно такая же ситуация. Например, сколько всего региональных телефонных кодов? Региональный телефонный код в США состоит из трех цифр. Если все эти коды будут задействованы (на самом деле пока не все, но мы это проигнорируем), получится 10^3, или 1000 таких кодов, — от 000 до 999. Сколько семизначных телефонных кодов может быть в зоне действия регионального кода 212? 10^7, или 10 000 000. Сколько телефонных номеров может быть в зоне действия регионального кода 212, причем с префиксом 260? 10^4, или 10 000.

Аналогично в двоичной системе количество возможных кодов всегда равно двойке в некоторой степени, где степень — количество битов.

Количество битов	Количество кодов
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

Каждый дополнительный бит удваивает количество кодов.

Если известно, сколько нужно кодов, можно ли рассчитать, сколько для этого потребуется битов? Иными словами, как считать в противоположном направлении по вышеприведенной таблице?

Используемый при этом метод иногда называется «*логарифм по основанию два*». Логарифм — феномен, противоположный возведению в степень. Известно, что 2 в седьмой степени равно 128. Логарифм 128 по основанию 2 равен 7. В более строгой математической нотации получается, что выражение

$$2^7 = 128$$

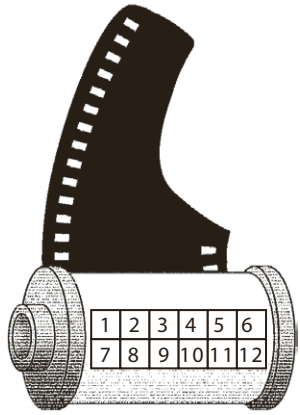
эквивалентно выражению

$$\log_2 128 = 7.$$

Итак, если логарифм 128 по основанию 2 равен 7, а логарифм 256 по основанию 2 равен 8, то каков логарифм 200 по основанию 2? На самом деле он равен примерно 7,64, но нам этого знать и не требуется.

Как правило, биты скрыты от поверхностного взгляда в глубинах электронных устройств. Вы не увидите их на компакт-диске, в электронных часах или внутри компьютера. Лишь иногда биты различимы вполне отчетливо.

Приведу пример. Если у вас есть обычный фотоаппарат с 35-миллиметровой пленкой, возьмите кассету и поверните ее, как показано на рисунке.



Перед вами предстанет набор серебристых и черных квадратов, отдаленно напоминающий шахматную доску. На рисунке я пронумеровал их от 1 до 12. Этот набор называется DX-кодировкой. Эти 12 квадратов в действительности представляют собой 12 бит. Серебристый квадрат соответствует 1, черный — 0. Квадраты 1 и 7 всегда серебристые (1).

Что значат эти биты? Вы, возможно, знаете, что пленки различаются по светочувствительности. Пленка высокой чувствительности считается «быстрой», поскольку на ней можно производить съемку с очень короткими экспозициями. Чувствительность пленки измеряется в единицах ASA (American Standards Association, Американская ассоциация по стандартам), причем наиболее популярны пленки с чувствительностью 100, 200 и 400 единиц*. Чувствительность в единицах ASA не только напечатана на упаковке, но и закодирована на кассете.

Различают 24 стандартные степени светочувствительности для фотопленки.

25	32	40
50	64	80
100	125	160
200	250	320
400	500	640
800	1000	1250
1600	2000	2500
3200	4000	5000

* Сегодня в большинстве стран мира используется система ISO (International Organization for Standartization, Международная организация по стандартизации). Обозначения светочувствительности пленки по системам ASA и ISO совпадают. *Прим. науч. ред.*

Сколько битов нужно, чтобы закодировать чувствительность пленки? Ответ прост: пять. Мы знаем, что $2^4 = 16$, а это слишком мало. А вот $2^5 = 32$ — больше чем достаточно.

Соответствие между квадратами на кассете и чувствительностью пленки показано в таблице.

Квадрат 2	Квадрат 3	Квадрат 4	Квадрат 5	Квадрат 6	Светочувствительность пленки
0	0	0	1	0	25
0	0	0	0	1	32
1	0	0	1	0	50
1	0	0	0	1	64
1	0	0	1	1	80
0	1	0	1	0	100
0	1	0	0	1	125
0	1	0	1	1	160
1	1	0	1	0	200
1	1	0	0	1	250
1	1	0	1	1	320
0	0	1	1	0	400
0	0	1	0	1	500
0	0	1	1	1	640
1	0	1	1	0	800
1	0	1	0	1	1000
1	0	1	1	1	1250
0	1	1	1	0	1600
0	1	1	0	1	2000
0	1	1	1	1	2500
1	1	1	1	0	3200
1	1	1	0	1	4000
1	1	1	1	1	5000

Эти коды используются в большинстве современных 35-миллиметровых фотоаппаратов. Если на вашем фотоаппарате выдержка или тип пленки устанавливаются вручную, эти коды в нем не применяются. Если же коды все-таки считываются, присмотритесь к фотоаппарату, когда будете вставлять пленку. Вы увидите шесть металлических контактов, соответствующих квадратам с первого по шестой на кассете. Серебристые квадраты — это просто открытая металлическая поверхность кассеты, которая является проводником. Черные квадраты покрыты краской, которая не проводит электричество.

Электрическая схема фотоаппарата построена так, что ток подводится к первому квадрату на кассете (он всегда серебристый). Этот ток будет (или не будет) проведен пятью контактами на квадратах со второго по шестой в зависимости от того, окрашены они изолирующей краской или нет. Так, если ток

присутствует на контактах 4 и 5, но отсутствует на контактах 2, 3 и 6, в фотоаппарат вставлена пленка 400 ASA. При съемке выдержка будет установлена автоматически.

В бюджетных фотоаппаратах считываются только квадраты 0 и 3, а чувствительность пленки считается равной 50, 100, 200 или 400 единицам ASA.

Квадраты с восьмого по двенадцатый в большинстве аппаратов также не используются. В квадратах 8, 9 и 10 зашифровано число кадров на пленке, а квадраты 11 и 12 содержат сведения о том, черно-белая или цветная пленка, позитивная или негативная.

Вероятно, чаще всего вам приходилось сталкиваться с двоичными числами в коде UPC (Universal Product Code, универсальный код продукта), или просто штрихкоде, — наборе черных полос, который сегодня присутствует практически на любой упаковке. Штрихкод — наглядный символ повсеместного проникновения компьютеров в нашу жизнь.

Хотя у некоторых людей штрихкод вызывает приступы паранойи, это совершенно безобидная вещь, изобретенная для автоматизации розничной торговли и учета товаров. Со своей задачей он справляется вполне успешно. Благодаря ему, например, современные кассовые аппараты выдают покупателю чек, в котором подробно расписаны все его покупки, чего без штрихкода сделать нельзя.

Нас же в первую очередь интересует, что код UPC является двоичным, хотя на первый взгляд этого не скажешь. Давайте разберемся, как устроен штрихкод и как он работает.

Чаще всего встречается штрихкод, состоящий из нескольких цифр и 30 вертикальных полосок различной толщины, разделенных пустыми интервалами переменной толщины. В качестве примера рассмотрим штрихкод, нанесенный на банку куриного супа с вермишелью фирмы Campbell.

Сразу хочется разделить код UPC на тонкие и жирные полоски, узкие и широкие промежутки, и это действительно помогло бы разобраться в его структуре. Черные полоски и пустые промежутки штрихкода бывают различной ширины (всего четыре полоски).

Конечно, удобнее трактовать UPC как набор битов. Имейте в виду, что сканирующему устройству нет нужды просматривать штрихкод целиком, тем более прибор не может интерпретировать цифры в его основании, поскольку это потребовало бы применения сложной компьютерной технологии распознавания символов (Optical Character Recognition, OCR). Сканеру достаточно «увидеть» тонкий срез штрих-кода. Код UPC делают таким большим просто для того, чтобы кассиру легче было нацелить на него сканер. Срез, попадающий в сканер, выглядит следующим образом.

Почти как азбука Морзе, правда?

Сканируя эту информацию слева направо, компьютер присваивает бит 1 первой встреченной черной полоске и бит 0 первому промежутку. Следующие промежутки и штрихи считываются как последовательности одного, двух, трех или четырех битов в зависимости от ширины штриха или промежутка. В битовом представлении этот штрихкод выглядит так.

```
10100011010110001001100100011010001101000110101010111001011001101101100100111011001101000100101
```

Итак, весь UPC — просто последовательность из 95 бит. В данном случае их можно сгруппировать.

Биты	Значение
101	Левый шаблон-ограничитель
0001101	Цифры слева
0110001	
0011001	
0001101	
0011001	
01010	Центральный шаблон-разделитель
1110010	Цифры справа
1100110	
1101100	
1001110	
1100110	
1000100	
101	Правый шаблон-ограничитель

Первые три бита — всегда 101. Они называются левым шаблоном-ограничителем и нужны для того, чтобы настроить сканирующее устройство. По шаблону-ограничителю сканер определяет ширину штриха и промежутка, соответствующую одному биту. Иначе на всех упаковках код UPC пришлось бы делать одинакового размера.

За левым шаблоном-ограничителем следует шесть групп по семь бит в каждой. В них закодированы десятичные цифры от 0 до 9, в чем мы убедимся чуть позже. Затем идет 5-битовый центральный шаблон-разделитель — фиксированная группа битов (всегда 01010), используемая как встроенная контрольная система. Не найдя центрального шаблона-разделителя в нужном месте, сканер считает штрихкод неверным. В частности, так выявляют плохо пропечатанные или поддельные штрихкоды.

За центральным шаблоном-разделителем всегда идут еще шесть групп по семь бит каждая, а за ними — правый шаблон-ограничитель, всегда равный 101. Позже я расскажу, почему благодаря наличию правого шаблона-ограничителя штрихкод можно сканировать и в обратном направлении, то есть справа налево.

Всего в коде UPC зашифровано 12 десятичных цифр. Шесть из них закодированы с его левой стороны, по семь бит в каждой. Для их расшифровки применяется таблица.

Левосторонние коды

0001101 = 0	0110001 = 5
0011001 = 1	0101111 = 6
0010011 = 2	0111011 = 7
0111101 = 3	0110111 = 8
0100011 = 4	0001011 = 9

Обратите внимание: каждый 7-битовый код начинается с 0 и заканчивается 1. Встретив 7-битовый код, который начинается с 1, а заканчивается 0, сканер «понимает», что код UPC либо неверно прочитан, либо подделан. Кроме того, в каждом коде группы единиц встречаются лишь дважды. Это значит, что каждая десятичная цифра в коде UPC зашифрована двумя вертикальными штрихами.

Еще одна особенность кодов в этой таблице — нечетное количество единиц в каждом из них. Она также позволяет проверить корректность штрихкода — так называемый контроль четности (parity). Группа битов обладает *четным паритетом*, если в ней четное количество битов-единиц, и *нечетным паритетом*, если в ней нечетное количество битов-единиц.

Для расшифровки битов в правой части штрихкода применяется таблица.

Правосторонние коды

1110010 = 0	1001110 = 5
1100110 = 1	1010000 = 6
1101100 = 2	1000100 = 7
1000010 = 3	1001000 = 8
1011100 = 4	1110100 = 9

Эти коды дополняют коды из предыдущей таблицы. Там, где в левосторонних кодах был 0, теперь стоит 1, и наоборот. Правосторонние коды всегда начинаются с 1 и заканчиваются 0. Кроме того, число битов 1 в них всегда четное, что можно применять для контроля четности. Вот мы и готовы к расшифровке UPC. С помощью двух приведенных выше таблиц можно определить 11 цифр, зашифрованных на банке Campbell Soup.

0 51000 01251 7

Какая досада! Да, это те самые цифры, что напечатаны под штрихкодом. На самом деле это очень удобно: если сканер по каким-то причинам не смог прочитать код, кассир может ввести его вручную. Вы наверняка видели, как это бывает. Конечно, получается, что весь наш труд по расшифровке штрихкода был напрасным, к тому же никакой секретной информации мы так и не получили: просто 30 вертикальных штрихов превратились в 12 цифр.

Первая цифра (в данном случае 0) характеризует тип кода.

0 означает, что перед нами обычный код UPC. Если код нанесен на упаковку с товаром переменного веса, например с мясом или овощами, он начинается с 2. Товары со скидкой обозначаются цифрой 5.

Следующие пять цифр — код производителя. В нашем примере код 51000 соответствует компании Campbell Soup. Он есть на всех продуктах марки Campbell. За ними следует пятизначный (01251) код конкретного продукта этой компании, в нашем случае код банки с куриным супом. Код продукта информативен лишь в сочетании с кодом производителя. У куриного супа с вермишелью, выпущенного другой компанией, будет другой код продукта, в свою очередь код 01251 может значить нечто совершенно иное у другого производителя.

Вопреки распространенному мнению, в код UPC не включается цена товара. Информация о ней извлекается из компьютерной базы данных, используемой в кассовых аппаратах наряду со сканерами.

Последняя цифра (здесь — 7) называется *символом проверки остатка* и тоже используется для исключения ошибок. Чтобы проверить его на практике, присвоим букву каждой из первых 11 цифр (наш пример 0 51000 01251).

A BCDEF GHIJK

Теперь вычислим:

$$3 \times (A + C + E + G + I + K) + (B + D + F + H + J).$$

Вычтем результат из ближайшего большего числа, кратного десяти. Полученное число и будет символом проверки остатка для куриного супа с вермишелью Campbell:

$$3 \times (0 + 1 + 0 + 0 + 2 + 1) + (5 + 0 + 0 + 1 + 5) = 3 \times 4 + 11 = 23.$$

Ближайшее большее число, кратное десяти, — 30. Значит, 30 – 23 = 7.

Это число — результат проверки остатка — напечатано под штрихкодом и зашифровано в нем. Такая проверка — одна из форм избыточности. Если остаток, вычисленный по штрихкоду, не совпадет с остатком, явно указанным в нем, штрихкод будет сочтен недействительным.

Как правило, для представления десятичной цифры от 0 до 9 достаточно четырех бит. В штрихкодах используется по семь бит на цифру. Целыми 95 бит закодировано всего 11 значимых десятичных цифр. Если учесть, что UPC с обеих сторон ограничен пустым пространством, эквивалентным девяти нулевым битам, получается, что во всем штрихкоде 11 цифр закодировано 113 бит, по 10 бит на цифру!

Такая избыточная надежность отчасти требуется для защиты от ошибок. Код товара был бы не слишком полезен, если бы покупатель мог в два счета подправить его фломастером.

Кроме того, UPC удобен, поскольку его можно считывать в обоих направлениях. Если в первых считанных цифрах количество единиц четно, сканер распознает, что код читается справа налево. Для расшифровки правосторонних цифр компьютер использует следующую таблицу.

Правосторонние коды в обратном направлении

0100111 = 0	0111001 = 5
0110011 = 1	0000101 = 6
0011011 = 2	0010001 = 7
0100001 = 3	0001001 = 8
0011101 = 4	0010111 = 9

Вот таблица левосторонних кодов.

Левосторонние коды в обратном направлении

1011000 = 0	1000110 = 5
1001100 = 1	1111010 = 6
1100100 = 2	1101110 = 7
1011110 = 3	1110110 = 8
1100010 = 4	1101000 = 9

Эти 7-битовые коды отличаются от кодов, считываемых слева направо. Никакой путаницы не возникает.

Наше знакомство с кодами началось с азбуки Морзе, состоящей из точек, тире и промежутков между ними. Азбука Морзе, на первый взгляд, имеет мало общего с нулями и единицами, на деле же сводится именно к ним.

Вспомните устройство азбуки Морзе. Тире втрое длиннее точки. Точки и тире в пределах одной буквы разделены паузами продолжительностью в одну точку. Промежутки между буквами по длительности равны одному тире. Слова разделяются паузами в два тире.

Чтобы немного упростить анализ, допустим, что длина тире превышает длину точки не в три, а в два раза. Это означает, что точка соответствует одному единичному биту, а тире — двум единичным битам.

Паузы состоят из нулевых битов.

Вот простейшая таблица с азбукой Морзе из главы1.

A	• —	H	• • • •	O	— — —	U	• • —
B	— • • •	I	• •	P	• — — •	V	• • • —
C	— • — •	J	• — — —	Q	— — • —	W	• — —
D	— • •	K	— • —	R	• — •	X	— • • —
E	•	L	• — • •	S	• • •	Y	— • — —
F	• • — •	M	— —	T	—	Z	— — • •
G	— — •	N	— •				

А вот та же таблица, преобразованная в биты.

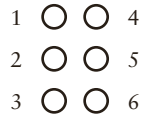
A	101100	H	101010100	O	1101101100	U	10101100
B	1101010100	I	10100	P	10110110100	V	1010101100
C	11010110100	J	101101101100	Q	110110101100	W	101101100
D	11010100	K	110101100	R	10110100	X	11010101100
E	100	L	1011010100	S	1010100	Y	110101101100
F	1010110100	M	1101100	T	1100	Z	11011010100
G	110110100	N	110100				

Обратите внимание: все коды начинаются с 1 и кончаются парой 0, представляющей паузу между буквами в пределах одного слова. Кодом пробела между словами является дополнительная пара 0. Таким образом, на азбуке Морзе фраза Hi there выглядит так.

Представив ее в битах, мы получим нечто очень похожее на срез штрихкода.

```
1010101001010000110010101010010010110100100000
```

В битовом выражении азбука Брайля гораздо проще азбуки Морзе. Шрифт Брайля является 6-битовым кодом. Каждый символ — набор из шести точек, каждая из которых может быть выпуклой или плоской. Как говорилось в главе 3, обычно точки нумеруются от 1 до 6.



Например, вот как записывается шрифтом Брайля слово code, где крайний левый бит соответствует первой позиции, а крайний правый — шестой.

Позже мы узнаем, что с помощью битов можно зашифровать не только коды товаров, чувствительность пленки, художественную ценность фильма, способ наступления британской армии или послание любимой женщине, но и любые слова, изображения, звуки, музыку и кино. В своей основе биты — это числа. Для представления информации в форме битов достаточно пересчитать количество доступных возможностей. Это количество определяет, сколько битов понадобится для того, чтобы присвоить каждой возможности уникальный номер.

Биты также играют важную роль в логике, находящейся на стыке философии и математики; ее главная цель — определение истинности или ложности некоего утверждения. Истину и ложь также можно обозначить через 1 и 0.

Глава 10
Логика и переключатели

Что есть истина? Аристотель полагал, что она как-то связана с логикой. Сборник его сочинений под названием «Органон» (датируемый IV веком до н. э.) — самое раннее произведение, где подробно освещается эта тема. Для древних греков логика — средство анализа языка с целью нахождения истины, поэтому она считалась формой философии. Основа логики Аристотеля — *силлогизм*. Самый известный силлогизм (который фактически отсутствует в работах Аристотеля) формулируется так:

Все люди смертны;
Сократ — человек;
следовательно, Сократ смертен.

В силлогизме из двух считающихся истинными предпосылок выводится заключение.

Смертность Сократа может показаться достаточно очевидной, однако существует множество разнообразных силлогизмов. Рассмотрим следующие две предпосылки, которые предложил математик XIX века Чарльз Доджсон, известный как Льюис Кэрролл:

Все философы логичны;
нелогичный человек всегда упрям.

В данном случае вывод не очевиден. Он формулируется так: «Некоторые упрямые люди не являются философами». Обратите внимание на неожиданное и привносящее неопределенность слово «некоторые».

На протяжении более двух тысяч лет математики боролись с логикой Аристотеля, пытаясь укротить ее с помощью математических символов и операторов. До XIX века ближе всех к решению этой задачи удалось подойти только Готфриду Вильгельму фон Лейбницу (1646–1716), который занимался

логикой в молодости, а затем заинтересовался иными вещами, например одновременно с Исааком Ньютоном разработал дифференциальное исчисление (независимо от него)*. Затем на сцену вышел Джордж Буль.

Джордж Буль родился в 1815 году в Англии в социуме, где его шансы на успех были очень малы. Поскольку он был сыном башмачника и бывшей горничной, его перспективы не сильно отличались от перспектив его предков по причине жесткой классовой иерархии британского общества. Однако благодаря своему пытливому уму и отцу, который интересовался наукой, математикой и литературой, молодой Джордж получил образование, как правило являющееся привилегией мальчиков из высших классов общества. Он изучал латынь, греческий язык и математику. Ранние работы Буля по математике позволили ему в 1849 году стать первым профессором математики в Королевском колледже Корка.

Несколько математиков в середине 1800-х годов работали над формальным определением логики (среди них особо выделялся Огастес де Морган). Однако именно Буль совершил настоящий концептуальный прорыв: сначала в короткой книге «Математический анализ логики, или Очерк исчисления дедуктивных умозаключений» (1847), затем в гораздо более объемном и амбициозном произведении «Исследование законов мышления, на которых основаны математические теории логики и вероятностей» (1854), которое кратко также называется «Исследование законов мышления». Буль умер в 1864 году в возрасте 49 лет от пневмонии, которую он подхватил, попав под дождь по дороге на лекцию.

Название книги Буля 1854 года говорит о постановке амбициозной задачи: поскольку мозг разумного человека мыслит, используя логику, то, найдя способ математического представления логики, мы получим математическое описание того, как работает мозг. Разумеется, в наше время такое ви́дение кажется весьма наивным (или просто оно значительно опережает свое время).

Изобретенная Булем алгебра очень похожа на обычную. В обычной алгебре *операнды* (обычно буквы) обозначают цифры, а *операторы* (например, «+» и «×») указывают, как эти числа должны объединяться. Как правило, мы используем обычную алгебру для решения таких задач: у Ани есть три яблока. У Бетти в два раза больше яблок, чем у Ани. У Кармен на пять яблок больше, чем у Бетти. У Дейрдре в три раза больше яблок, чем у Кармен. Сколько яблок у Дейрдре?

---

* Лейбниц и Ньютон предложили разный терминологический аппарат и различные формы записи дифференциального исчисления. В итоге победил вариант Лейбница. *Прим. науч. ред.*

Чтобы решить эту задачу, сначала преобразуем ее в арифметические выражения, используя четыре буквы, соответствующие количеству яблок, имеющихся у каждой из четырех женщин:

$$
\begin{aligned}
&А = 3;\\
&Б = 2 \times А;\\
&К = Б + 5;\\
&Д = 3 \times К.
\end{aligned}
$$

Мы можем объединить эти четыре выражения в одно путем подстановки, а затем уже выполнить операции сложения и умножения:

$$
\begin{aligned}
&Д = 3 \times К;\\
&Д = 3 \times (Б + 5);\\
&Д = 3 \times ((2 \times А) + 5);\\
&Д = 3 \times ((2 \times 3) + 5);\\
&Д = 33.
\end{aligned}
$$

Имея дело с обычной алгеброй, мы следуем определенным правилам. Эти правила настолько укоренились в практике, что мы больше не думаем о них как о правилах и даже иногда забываем их названия. Однако любая форма математики подчиняется им.

Первое правило заключается в том, что сложение и умножение являются *коммутативными* операциями. Это означает, что можно менять операнды местами в обеих частях выражения, не влияя на результат:

$$
\begin{aligned}
&А + В = В + А;\\
&А \times В = В \times А.
\end{aligned}
$$

Напротив, операции вычитания и деления *не являются* коммутативными.

Сложение и умножение — *ассоциативные* операции, то есть:

$$
\begin{aligned}
&А + (В + С) = (А + В) + С;\\
&А \times (В \times С) = (А \times В) \times С.
\end{aligned}
$$

Наконец, умножение *дистрибутивно* по отношению к сложению:

$$А \times (В + С) = (А \times В) + (А \times С).$$

Другой характеристикой обычной алгебры является то, что она всегда оперирует числами, например килограммами сыра, количеством уток, расстоянием, которое прошел поезд, или возрастом членов семьи. Гений Буля сделал алгебру более абстрактной, отделив ее от концепции числа. В булевой алгебре (именно такое название получила алгебра Буля) операнды относятся не к числам, а к *классам*. Класс — это просто набор предметов, который в дальнейшем стал *множеством*.

Поговорим о кошках. Кошки могут быть мужского и женского пола. Для удобства множество котов будем обозначать буквой *М*, а множество кошек — *Ж*. Имейте в виду, что эти два символа не соответствуют количеству кошек. Количество котов и кошек может меняться с течением времени по мере того, как новые особи рождаются, а старые, к сожалению, уходят в мир иной. Эти буквы обозначают классы кошек со специфическими характеристиками. Говоря о котах, мы можем просто сказать «М».

Мы также можем использовать другие буквы для обозначения окраса кошек: буквой *Р* описать множество рыжих, буквой *Ч* — множество черных, буквой *Б* — множество белых, а буквой *Д* — множество кошек всех «других» цветов, то есть кошек, не входящих в классы Р, Ч или Б.

Наконец (по крайней мере, в нашем примере) кошки могут быть либо стерилизованными, либо нет. Давайте обозначим буквой *С* множество стерилизованных кошек, а буквой *Н* — множество нестерилизованных.

В обычной (числовой) алгебре операторы «+» и «×» используются для обозначения операций сложения и умножения. В булевой алгебре применяются те же символы «+» и «×», что может вызвать путаницу. Всем известно, как складывать и умножать числа в обычной алгебре, но как можно складывать и умножать *классы*?

Дело в том, что в булевой алгебре мы фактически ничего не складываем и не умножаем. Вместо этого символы «+» и «×» означают нечто совершенно иное.

В булевой алгебре символ «+» — это *объединение* двух классов, которое предполагает объединение всего, относящегося к первому классу, со всем, относящимся ко второму. Например, выражение Ч + Б означает множество всех кошек черного и белого окраса.

Символ «×» — это *пересечение* двух классов, то есть пересечение множества элементов, принадлежащих как первому, так и второму классу. Например, Ж × Р — класс всех кошек женского пола и рыжего окраса. Как и в обычной алгебре, мы можем написать Ж × Р в виде Ж и Р или просто ЖР (именно так предпочитал писать сам Буль). Вы можете рассматривать эти две буквы в качестве двух прилагательных, описывающих множество «рыжие кошки женского пола».

Чтобы не спутать обычную алгебру с булевой, вместо символов «+» и «×» для обозначения объединения и пересечения классов иногда используются символы ∪ и ∩.

Однако освобождающее влияние Буля на математику отчасти заключалось в том, чтобы сделать использование знакомых операторов более абстрактным, поэтому, следуя его примеру, я решил не вводить новые символы.

Коммутативные, ассоциативные и дистрибутивные правила остаются справедливыми в булевой алгебре. Более того, здесь оператор «+» является дистрибутивным по отношению к оператору «×», чего нельзя сказать об обычной алгебре:

$$Б + (Ч \times Ж) = (Б + Ч) \times (Б + Ж).$$

Объединение белых и черных кошек-самок равнозначно пересечению двух объединений: белых и черных кошек, а также белых кошек и кошек-самок. Это сложно понять, но все именно так и устроено.

Булевой алгебре необходимы еще два символа. Они смахивают на числа, но ими не являются, поскольку иногда с ними обращаются не так, как с числами. Символ 1 означает множество всех вещей, о которых мы говорим. В данном примере 1 — это множество всех кошек:

$$М + Ж = 1.$$

Значит, множество всех кошек содержит самцов и самок. Точно так же оно включает всех кошек рыжего, черного, белого и других окрасов:

$$Р + Ч + Б + Д = 1.$$

Кроме того, множество всех кошек можно получить и так:

$$С + Н = 1.$$

Символ 1 может использоваться со знаком минус, чтобы указать на множество всех вещей, *исключающее* некое подмножество, например:

$$1 - М.$$

Как видите, это множество всех кошек, кроме самцов. Множество всех кошек, исключающее всех самцов, соответствует множеству кошек женского пола:

$$1 - М = Ж.$$

Другой необходимый символ — 0, а в булевой алгебре 0 означает пустое множество, которое ничего не содержит. Пустое множество — результат пересечения двух взаимоисключающих множеств, например множество кошек-гермафродитов:

$$Ж \times М = 0.$$

Обратите внимание: символы 1 и 0 иногда работают одинаково в булевой и в обычной алгебре. Например, пересечение множества всех кошек и кошек женского пола соответствует множеству кошек-самок:

$$1 \times Ж = Ж.$$

Пересечение пустого множества и множества кошек-самок представляет пустое множество:

$$0 \times Ж = 0.$$

Объединение пустого множества и множества всех кошек-самок — это множество кошек-самок:

$$0 + Ж = Ж.$$

Однако иногда результаты в булевой и в обычной алгебре отличаются. Например, объединение всех кошек и кошек-самок соответствует множеству всех кошек:

$$1 + Ж = 1.$$

Это не имеет смысла в обычной алгебре.

Поскольку Ж — множество всех кошек-самок, а 1 – Ж — множество всех кошек, которые не являются самками, объединение этих двух множеств соответствует 1:

$$Ж + (1 - Ж) = Ж + М = 1.$$

Пересечение двух множеств соответствует 0:

$$Ж \times (1 - Ж) = 0.$$

С исторической точки зрения эта формулировка — важная веха в логике, называемая *законом противоречия*, который гласит, что нечто не может одновременно являться собой и своей противоположностью.

Где булева алгебра действительно отличается от обычной, так это в следующем выражении:

$$Ж \times Ж = Ж.$$

Пересечение множества кошек-самок и множества кошек-самок по-прежнему множество кошек-самок. Это выражение имеет смысл в булевой алгебре. Однако оно неверное, если бы буква *Ж* означала число. Буль считал, что выражение $Х^2 = Х$ является единственным выражением, отличающим его алгебру от обычной. Вот еще одно булево выражение, которое выглядит странно с точки зрения обычной алгебры:

$$Ж + Ж = Ж.$$

Объединение множества кошек-самок и множества кошек-самок по-прежнему является множеством кошек-самок.

Булева алгебра предоставляет математический метод для решения силлогизма Аристотеля. Давайте рассмотрим первые две его части:

Все люди смертны;
Сократ — человек.

Буквой *Л* мы обозначим множество всех людей, буквой *Х* — множество всех смертных существ, а буквой *С* — множество Сократов. Что означает выражение «все люди смертны»? Пересечение множества всех людей и множества всех смертных существ — это множество всех людей:

$$Л \times Х = Л.$$

Выражение $Л \times Х = Х$ было бы неправильным, поскольку множество всех смертных существ включает кошек, собак и деревья.

Выражение «Сократ — человек» означает, что пересечение множества Сократов (очень небольшого множества) и множества всех людей (гораздо более крупного множества) представляет множество Сократов:

$$С \times Л = С.$$

Поскольку из первого уравнения известно, что Л равно Л × Х, можем подставить это выражение во второе:

$$С \times (Л \times Х) = С.$$

Согласно ассоциативному закону это равнозначно выражению:

$$(С \times Л) \times Х = С.$$

Однако мы уже знаем, что С × Л равно С, поэтому можем упростить выражение, используя эту подстановку:

$$С \times Х = С.$$

Теперь мы закончили. Эта формула указывает, что пересечение множества Сократов и множества всех смертных существ есть С, а это значит, что Сократ смертен. Если бы вместо этого оказалось, что С × Х равно 0, мы бы пришли к выводу, что Сократ не был смертным. Если бы мы обнаружили, что С × Х равно Х, то вывод заключался бы в том, что Сократ является единственным смертным существом, а все остальные бессмертны.

Использование булевой алгебры может показаться излишним для доказательства очевидного факта (особенно учитывая то, что Сократ доказал собственную смертность 2400 лет назад), однако ее можно использовать для того, чтобы определить, удовлетворяет ли что-то определенному набору критериев. Возможно, однажды вы зайдете в зоомагазин и скажете продавцу: «Мне нужен стерилизованный кот белого или рыжего окраса, или стерилизованная кошка любого окраса, кроме белого, или я возьму любую из имеющихся у вас черных кошек». И продавец скажет, что вам нужна кошка из множества, описываемого следующим выражением:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$

Верно? И вы ответите: «Да! Точно!»

Проверяя, правильно ли продавец вас понял, можно отказаться от понятий объединения и пересечения, вместо них использовать слова ИЛИ и И. Я пишу эти слова заглавными буквами, потому что они не только соответствуют понятиям в обычном языке, но и могут представлять собой операции в булевой алгебре. Когда вы формируете объединение двух множеств, вы фактически берете элементы из первого ИЛИ второго множества. А когда вы

формируете пересечение, то берете только те элементы, которые одновременно принадлежат первому И второму множествам. Кроме того, вы можете использовать слово НЕ везде, где встречается символ 1, за которым следует знак «минус». Таким образом:

— символ «+» (ранее обозначавший объединение) теперь означает ИЛИ;
— символ «×» (ранее обозначавший пересечение) теперь означает И;
— выражение «1 –» (ранее обозначавшее множество всех элементов, за исключением чего-то) теперь означает НЕ.

Именно поэтому приведенное выше выражение также может быть записано:

(М И С И (Б ИЛИ Р)) ИЛИ (Ж И С И (НЕ Б)) ИЛИ Ч.

Как видите, это соответствует тому, что вы сказали. Обратите внимание, как скобки уточняют ваши пожелания. Вам нужна кошка, принадлежащая одному из трех множеств.

(М И С И (Б ИЛИ Р))
ИЛИ
(Ж И С И (НЕ Б))
ИЛИ
Ч

С помощью этой формулы продавец может выполнить то, что называется *проверкой условия*. Незаметно мы перешли к несколько иной форме булевой алгебры, в которой буквы не обозначают множества. Вместо этого буквы теперь могут соответствовать числам. Однако буквам может быть присвоено только значение 0 или 1. Число 1 означает «да», «истина», данная конкретная кошка удовлетворяет этим критериям, число 0 — «нет», «ложь», данная кошка не удовлетворяет этим критериям.

Сначала продавец приносит нестерилизованного рыжего кота. Вот выражение, описывающее множество приемлемых кошек:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$

Вот как оно выглядит после подстановки значений 0 и 1:

$$(1 \times 0 \times (0 + 1)) + (0 \times 0 \times (1 - 0)) + 0.$$

Обратите внимание: единственными символами, которым было присвоено значение 1, являются M и P, поскольку речь идет о рыжем коте.

Теперь нужно упростить данное выражение. Если оно упрощается до 1, то кошка удовлетворяет вашим критериям; если оно упрощается до 0, то кошка критериям не удовлетворяет. Имейте в виду, что в процессе упрощения выражения мы на самом деле ничего не складываем и не умножаем, хотя обычно можем сделать вид, что выполняем эти операции. Большинство тех же правил применяются тогда, когда символ «+» означает ИЛИ, а символ «×» — И. Иногда в современных текстах для обозначения И и ИЛИ используются символы «∧» и «∨» вместо «×» и «+». Однако именно здесь символы «+» и «×», вероятно, имеют наибольший смысл.

Когда символ «×» означает И, возможны результаты:

$$0 \times 0 = 0;$$
$$0 \times 1 = 0;$$
$$1 \times 0 = 0;$$
$$1 \times 1 = 1.$$

Другими словами, результат равен 1 только в том случае, если левый И правый операнды равны 1. Эта операция соответствует обычному умножению и называется *конъюнкцией*, и ее можно описать с помощью небольшой таблицы, аналогичной таблицам сложения и умножения, приведенным в главе 8.

И	**0**	**1**
0	0	0
1	0	1

Когда символ «+» означает ИЛИ, возможны следующие результаты.

$$0 + 0 = 0;$$
$$0 + 1 = 1;$$
$$1 + 0 = 1;$$
$$1 + 1 = 1.$$

Результат равен 1, если левый ИЛИ правый операнд равен 1. Исход этой операции похож на результаты обычного сложения, за исключением того, что

в данном случае 1 + 1 равно 1. Результаты операции ИЛИ, которая называется *дизъюнкцией,* можно представить в виде другой таблицы.

ИЛИ	**0**	**1**
0	0	1
1	1	1

Мы готовы использовать эти таблицы для вычисления:

$$(1 \times 0 \times 1) + (0 \times 0 \times 1) + 0 = 0 + 0 + 0 = 0.$$

Результат 0 — «нет», «ложь», этот котенок не подходит.

Затем продавец приносит стерилизованную белую кошку. Исходное выражение выглядело так:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$

Снова подставим в него значения 0 и 1:

$$(0 \times 1 \times (1 + 0)) + (1 \times 1 \times (1 - 1)) + 0.$$

И упростим его:

$$(0 \times 1 \times 1) + (1 \times 1 \times 0) + 0 = 0 + 0 + 0 = 0.$$

Еще один несчастный котенок отвергнут.

Затем продавец приносит стерилизованную серую кошку. (Серый соответствует критерию «другой окрас», то есть не белый, не черный и не рыжий.) Вот соответствующее выражение:

$$(0 \times 1 \times (0 + 0)) + (1 \times 1 \times (1 - 0)) + 0.$$

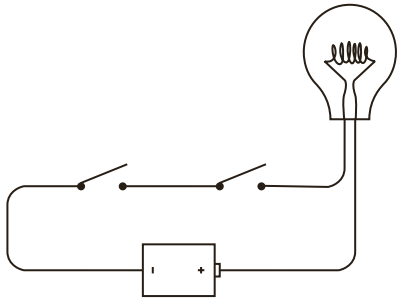
Теперь упростим его:

$$(0 \times 1 \times 0) + (1 \times 1 \times 1) + 0 = 0 + 1 + 0 = 1.$$

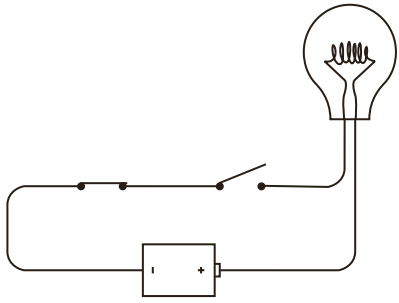
Результат вычисления, равный 1, означает «да», «истина», котенок нашел свой дом. (Кроме того, он оказался самым милым!)

Позже в тот же вечер, пока котенок спит у вас на коленях, вы спрашиваете себя, нельзя ли подключить несколько переключателей к лампочке для облегчения процесса проверки котят на соответствие вашим критериям. (Да, вы весьма странный ребенок.) Сами не зная того, вы вплотную приблизились к решающему концептуальному прорыву. Вы вот-вот проведете некоторые эксперименты, которые объединят алгебру Джорджа Буля с электричеством и сделают возможным проектирование и сборку компьютеров, работающих с двоичными числами. Однако пусть вас это не пугает.

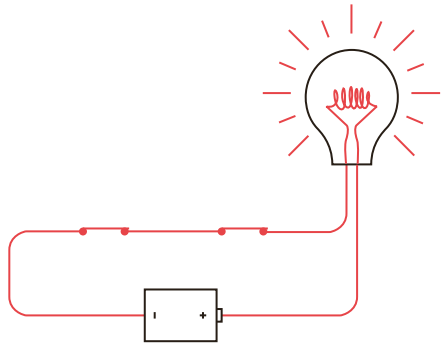
Чтобы поставить такой эксперимент, вы, как обычно, соединяете лампочку и батарейку, но используете два переключателя вместо одного.



Считается, что переключатели, подключенные друг за другом, соединены *последовательно*. Если вы замыкаете левый переключатель, ничего не происходит.



Если вы оставляете левый переключатель разомкнутым, а замыкаете правый, также ничего не произойдет. Лампочка загорается, когда и левый, и правый переключатели оказываются замкнутыми.



Ключевым в данном случае является союз «и». Левый *и* правый переключатели должны быть замкнуты, чтобы ток шел по цепи.

Эта схема решает небольшую логическую задачу. Фактически лампочка отвечает на вопрос: «Замкнуты ли оба переключателя?» Мы можем суммировать результаты работы этой схемы в следующей таблице.

Левый переключатель	**Правый переключатель**	**Лампочка**
Разомкнут	Разомкнут	Не горит
Разомкнут	Замкнут	Не горит
Замкнут	Разомкнут	Не горит
Замкнут	Замкнут	Горит

В предыдущей главе мы говорили о том, как с помощью двоичных цифр, или битов, можно представить любую информацию, начиная от чисел и заканчивая направлением большого пальца Роджера Эберта. Мы могли сказать, что ноль бит означает, что палец направлен вниз, а один бит — что палец направлен вверх. Переключатель может находиться в двух положениях, поэтому для его описания достаточно одного бита. Можно сказать, что 0 — это «переключатель разомкнут», а 1 — «переключатель замкнут». Лампочка также имеет два состояния, следовательно, для их описания достаточно одного бита. Можно сказать, что 0 — «лампочка не горит», а 1 — «лампочка горит». Теперь мы просто переписываем приведенную выше таблицу.

Левый переключатель	**Правый переключатель**	**Лампочка**
0	0	0
0	1	0
1	0	0
1	1	1

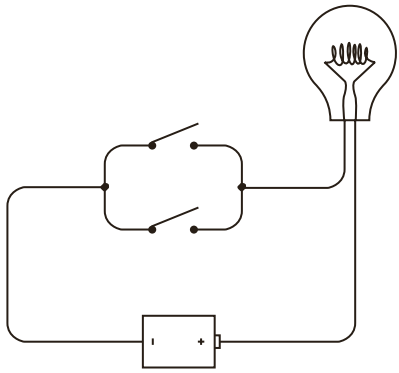
Обратите внимание: если мы поменяем местами левый и правый переключатели, результаты останутся прежними. Нам не обязательно различать переключатели. Именно поэтому таблицу можно переписать так, чтобы она напоминала приведенные И/ИЛИ.

Последовательное соединение переключателей	**0**	**1**
0	0	0
1	0	1

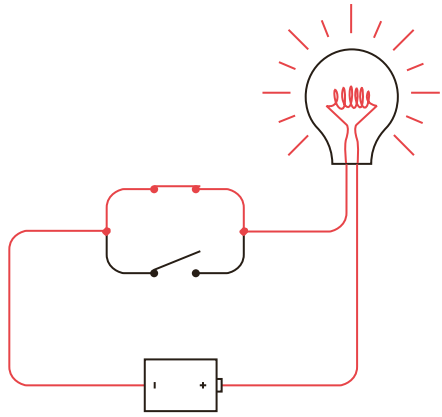
Действительно, это соответствует таблице с результатами выполнения булевой операции И.

И	**0**	**1**
0	0	0
1	0	1

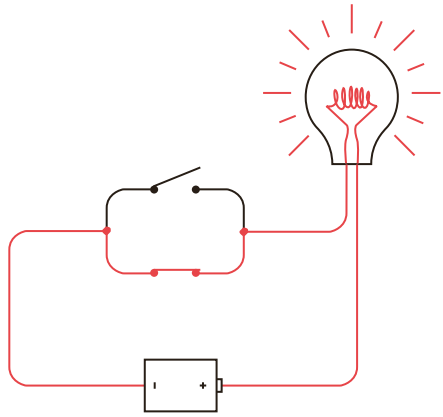
Эта простая схема фактически выполняет операцию И в булевой алгебре. Теперь попробуйте соединить два переключателя иначе.



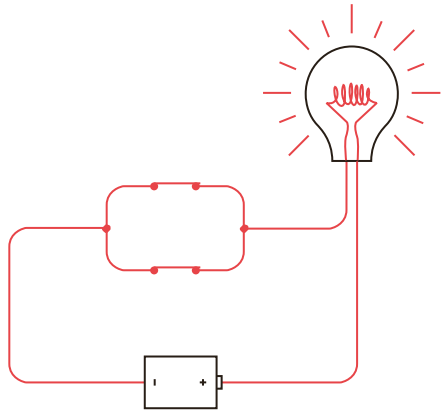
Переключатели соединены *параллельно*. Разница между этим и предыдущим способом соединения заключается в том, что эта лампочка загорится, если вы замкнете верхний переключатель.



Или нижний переключатель.



Можно также замкнуть оба переключателя.



Лампочка загорается, если замкнуть верхний *или* нижний переключатель. Ключевым словом в данном случае является союз «или».

Опять же, данная схема решает логическую задачу. Лампочка отвечает на вопрос: «Замкнут ли хотя бы один переключатель?» В следующей таблице показаны результаты работы этой схемы.

Левый переключатель	**Правый переключатель**	**Лампочка**
Разомкнут	Разомкнут	Не горит
Разомкнут	Замкнут	Горит
Замкнут	Разомкнут	Горит
Замкнут	Замкнут	Горит

Теперь снова используем 0 для обозначения разомкнутого переключателя или негорящей лампочки и 1 — для обозначения замкнутого переключателя или горящей лампочки, в результате чего получим следующую таблицу.

Левый переключатель	**Правый переключатель**	**Лампочка**
0	0	0
0	1	1
1	0	1
1	1	1

Опять же ничего не изменится, если переключатели поменять местами, поэтому таблицу можно заполнить следующим образом.

Параллельное соединение переключателей	**0**	**1**
0	0	1
1	1	1

Вероятно, вы уже догадались, что эта таблица соответствует результатам булевой операции ИЛИ.

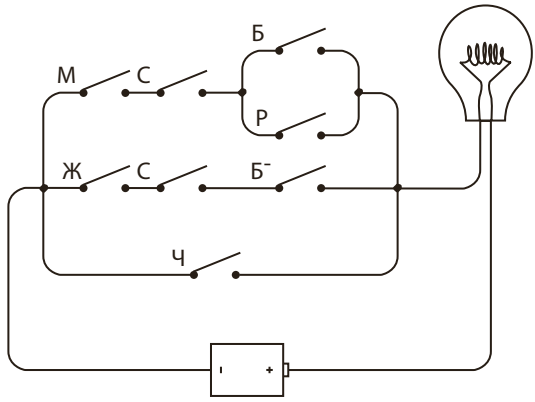
ИЛИ	**0**	**1**
0	0	1
1	1	1

Значит, два соединенных параллельно переключателя выполняют операцию, эквивалентную булевой операции ИЛИ.

Явившись в зоомагазин, вы сказали продавцу: «Мне нужен стерилизованный кот белого или рыжего окраса; или стерилизованная кошка любого окраса, кроме белого; или я возьму любую из имеющихся у вас черных кошек», — и продавец составил такое выражение:

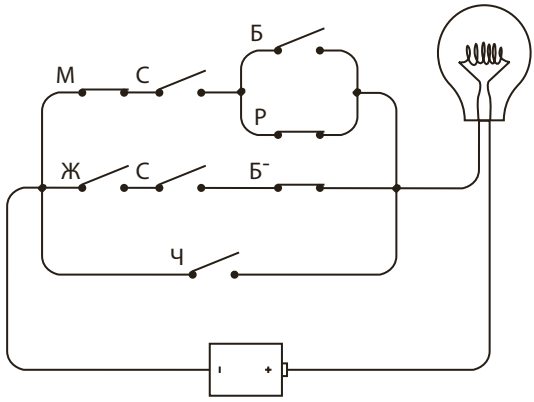
$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$

Теперь, когда вы знаете, что два соединенных последовательно переключателя выполняют логическую операцию И (обозначаемую символом «×»), а два переключателя, соединенных параллельно, — логическую операцию ИЛИ (обозначаемую символом «+»), вы можете соединить восемь переключателей.

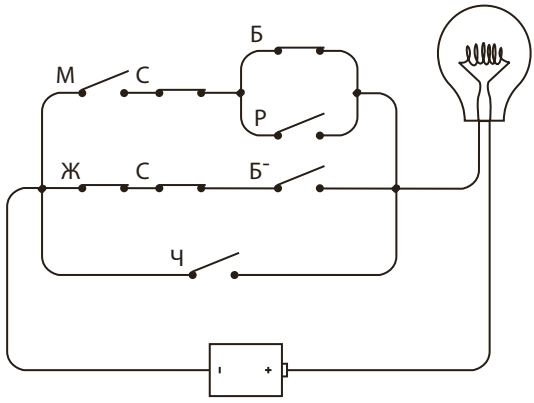


Все переключатели в этой схеме обозначены буквами, соответствующими буквам в булевом выражении. (Б⁻ означает НЕ Б и является альтернативным способом записи выражения 1 – Б). Действительно, если вы просмотрите электрическую схему слева направо и сверху вниз, то столкнетесь с буквами в том же порядке, в каком они представлены в выражении. Каждый символ «×» соответствует месту схемы, где два переключателя (или две группы переключателей) соединены последовательно, каждый символ «+» — месту схемы, в котором два переключателя (или две группы переключателей) соединены параллельно.

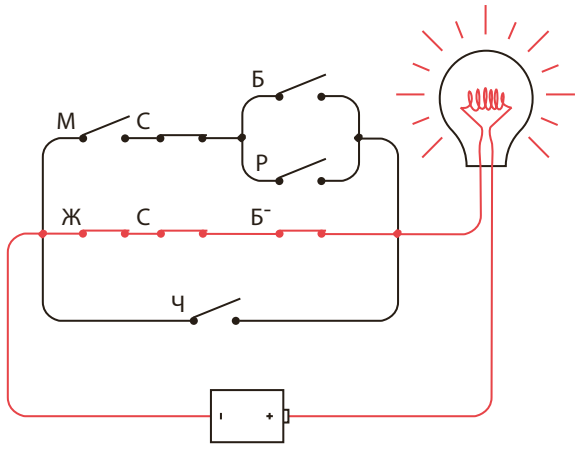
Как вы помните, продавец сначала принес нестерилизованного рыжего кота. Замкните соответствующие переключатели.



Несмотря на то что переключатели М, Р и НЕ Б замкнуты, лампочка не загорается. Затем продавец принес стерилизованную белую кошку.



Опять же, замкнуты не все нужные переключатели для того, чтобы загорелась лампочка. Наконец продавец приносит стерилизованную серую кошку.



Так можно замкнуть все нужные переключатели, зажечь лампочку и показать, что котенок удовлетворяет всем вашим критериям.

Джордж Буль никогда не собирал такую схему. Ему никогда не доводилось видеть логическое выражение, реализованное с помощью переключателей, проводов и лампочек. Разумеется, одним из препятствий было то, что лампа накаливания была изобретена только спустя 15 лет после смерти Буля. Однако Сэмюэл Морзе продемонстрировал свой телеграф в 1844 году — за десять лет до публикации книги Буля «Исследование законов мышления», и ему ничего не стоило заменить лампочки в приведенной выше схеме клопфером.

Никому в XIX веке не удалось уловить связь между булевыми операциями И и ИЛИ и последовательным и параллельным соединением простых переключателей — ни математику, ни электрику, ни оператору телеграфа*. Это не пришло в голову даже отцу-основателю компьютерной революции — Чарльзу Бэббиджу (1792–1871), который переписывался с Булем и был знаком с его работой, а большую часть жизни потратил на разработку разностной, а затем аналитической машины, которая спустя столетие будет считаться предшественником современных компьютеров. Сейчас мы знаем, что Бэббиджу помогло осознание того, что вместо шестеренок и рычагов для выполнения вычислений лучше использовать телеграфные реле.

Да, телеграфные реле.

---

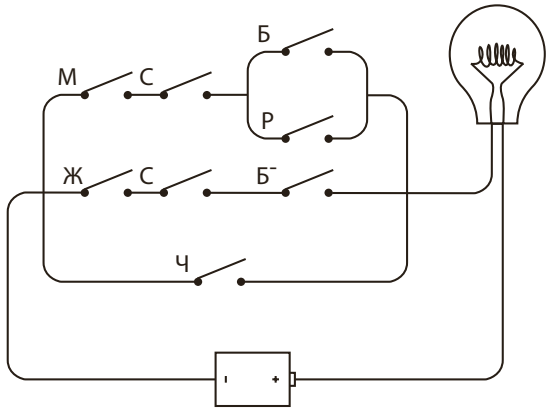
* То, что мы привыкли называть булевой алгеброй, булевыми (или булевскими) переменными — всего лишь побочный эффект прорыва, который совершил Буль. В тот момент, когда на сцене появился Буль, логика и математика развивались в значительной степени автономно на протяжении более двух тысяч лет! Буль при помощи своей алгебры показал, как можно соединить эти две дисциплины, и тем самым создал новую — математическую — логику. *Прим. науч. ред.*

Глава 11
Логические вентили

В далеком будущем, когда история примитивных вычислений XX века превратится в предания, кто-то, вероятно, предположит, что *логические вентили* были названы в честь одноименного сантехнического устройства. Это не совсем так. Мы вскоре увидим, что логические вентили действительно напоминают обычные вентили, через которые проходит вода, и выполняют элементарные логические задачи, блокируя или пропуская электрический ток.

В предыдущей главе мы рассматривали сценарий, когда вы вошли в зоомагазин и сказали продавцу: «Мне нужен белый или рыжий стерилизованный кот или стерилизованная кошка любого цвета, кроме белого, или любой кот или кошка черного цвета». Эти критерии можно объединить в следующее логическое выражение, а также выразить с помощью схемы из переключателей и лампочки:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$



Такая схема иногда называется *сетью*, хотя в настоящее время это слово гораздо чаще используется для обозначения соединенных между собой компьютеров, а не набора простых переключателей.

Несмотря на то что все обозначенные на схеме элементы были изобретены в XIX веке, тогда никто не представлял, что логические выражения можно реализовать непосредственно в виде электрических цепей. Эта возможность была осознана только в 1930-х годах Клодом Шенноном (1916–2001), который в 1938 году защитил знаменитую магистерскую диссертацию под названием «Символьный анализ реле и коммутаторов». Спустя десять лет впервые была опубликована его статья «Математическая теория связи», в которой слово «бит» (bit) использовалось для обозначения *двоичной цифры*.

Разумеется, задолго до 1938 года было известно, что для протекания тока при последовательном соединении двух переключателей оба должны быть замкнуты, а при параллельном соединении — лишь один из них. Однако никто так ясно и убедительно, как Шеннон, не показал, что для проектирования схем с переключателями инженеры-электрики могут использовать все инструменты булевой алгебры. В частности, если вы можете упростить логическое выражение, описывающее схему, то можете упростить и саму схему.

Например, выражение, содержащее ваши критерии выбора кошки, выглядит так:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч.$$

Используя сочетательный закон, мы можем изменить порядок переменных, объединенных знаком И («×»), и переписать выражение:

$$(С \times М \times (Б + Р)) + (С \times Ж \times (1 - Б)) + Ч.$$

Для ясности введу два дополнительных символа X и Y:

$$X = М \times (Б + Р);$$
$$Y = Ж \times (1 - Б).$$

Теперь выражение с критериями выбора кошки можно записать так:

$$(С \times X) + (С \times Y) + Ч.$$

Наконец, мы можем вернуть значения выражений, соответствующих символам X и Y.

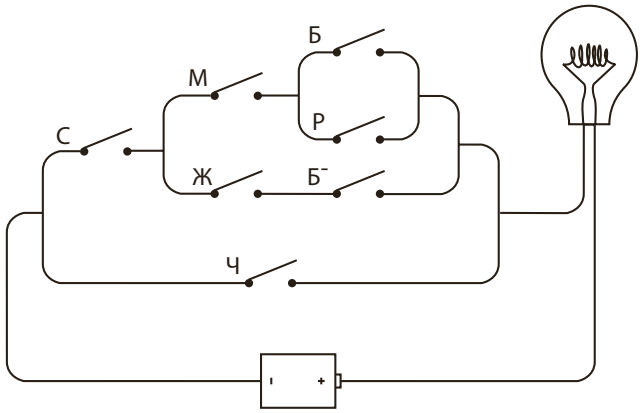
Обратите внимание: переменная С встречается в выражении дважды. Используя распределительный закон, это выражение можно переписать только с одной переменной С:

$$(С \times (Х + У)) + Ч.$$

Теперь подставим в выражение значения X и Y:

$$(С \times ((М \times (Б + Р)) + (Ж \times (1 - Б)))) + Ч.$$

Из-за множества скобок это выражение не выглядит упрощенным. Однако оно содержит на одну переменную меньше, а значит, в схеме меньше переключателей. Вот ее пересмотренная версия.



Действительно, увидеть, что эта схема эквивалентна предыдущей, легче, чем заметить тождество выражений.

На самом деле в этой цепи по-прежнему на три переключателя больше, чем нужно. Теоретически для выбора идеальной кошки должно быть достаточно четырех переключателей. Почему четырех? Каждый переключатель — это бит. Одного переключателя хватит для указания пола (разомкнутый — соответствует коту, замкнутый — кошке), еще один будет указывать на стерилизованную кошку в замкнутом состоянии и нестерилизованную — в разомкнутом, еще два позволят распознать цвет. Существуют четыре возможных цвета: белый, черный, рыжий и «другой». И мы знаем, что четыре варианта можно определить с помощью двух битов, поэтому для указания цвета нужно всего два переключателя. Например, белому цвету могут соответствовать два разомкнутых переключателя, черному — один замкнутый, рыжему — второй замкнутый, а «другим» — два замкнутых.

Теперь давайте построим пульт управления для выбора кошки, который будет состоять из лампочки и четырех переключателей (похожих на те, с помощью которых вы включаете и выключаете свет).

Переключатель замкнут, когда находится в положении вверх, разомкнут — когда находится в положении вниз. Боюсь, что обозначения двух переключателей для выбора цвета кошки могут показаться немного непонятными, однако это следствие попытки обойтись при создании пульта управления минимумом средств. Левый переключатель в этой паре обозначен буквой *Ч*; замыкание только левого переключателя (как показано на рисунке) соответствует черному цвету. Правый переключатель в этой паре обозначен буквой *Р*; замыкание только правого переключателя соответствует рыжему цвету, замыкание обоих — «другому» цвету (этот вариант обозначен буквой *Д*). Размыкание обоих переключателей соответствует белому цвету и обозначается буквой *Б* внизу.

Если пользоваться компьютерной терминологией, набор переключателей — это *устройство ввода* информации, управляющей поведением цепи. В данном случае переключатели соответствуют четырем битам, позволяющим описать кошку. *Устройством вывода* является лампочка, которая загорается, если положение переключателей согласуется с описанием подходящей кошки. Переключатели, изображенные на предыдущем рисунке, описывают нестерилизованную черную кошку. Ее характеристики удовлетворяют вашим критериям, поэтому лампочка загорается.

Теперь нам нужно лишь сконструировать схему, которая оживит этот пульт управления.

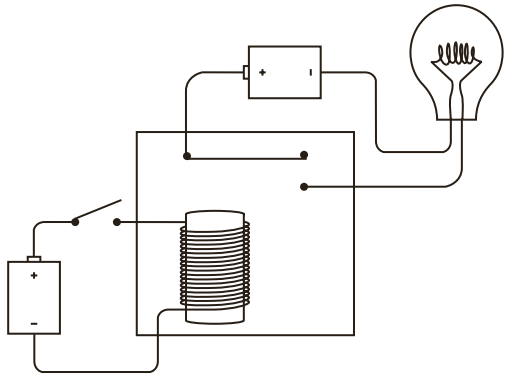
Как вы помните, диссертация Клода Шеннона называлась «Символьный анализ реле и коммутаторов». Описанные им реле были очень похожи на телеграфные, о которых мы говорили в главе 6. Однако к моменту публикации работы Шеннона реле использовались для других целей, в частности в телефонной сети.

Подобно переключателям, реле можно соединять последовательно и параллельно для решения простых логических задач. Эти комбинации называются *логическими вентилями*. Когда я говорю, что эти логические вентили

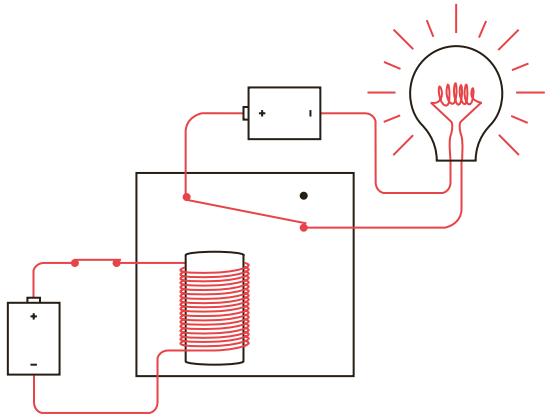
решают *простые* логические задачи, я имею в виду максимально простые задачи. Преимущество реле по сравнению с переключателями заключается в том, что их можно включать и выключать автоматически (с помощью других реле), а не вручную. Таким образом, логические вентили можно комбинировать для решения более сложных задач, например для выполнения простых арифметических операций. В следующей главе будет показано, как из переключателей, лампочек, источника питания и телеграфных реле можно собрать счетную машину (пусть и работающую исключительно с двоичными числами).

Как известно, реле играли ключевую роль в работе телеграфной системы. Из-за больших расстояний провода, соединяющие телеграфные станции, имели очень высокое сопротивление. Нужно было устройство, способное принимать слабый сигнал и передавать идентичный, но более мощный. Реле решало это задачу, используя электромагнит для управления переключателем. По сути, реле *усиливало* слабый сигнал для получения более мощного.

В наши планы не входит использование реле для усиления слабого сигнала. Нас интересует только то, что реле является переключателем, которым можно управлять не вручную, а с помощью электричества. Мы можем соединить реле с переключателем, лампочкой и парой батареек.



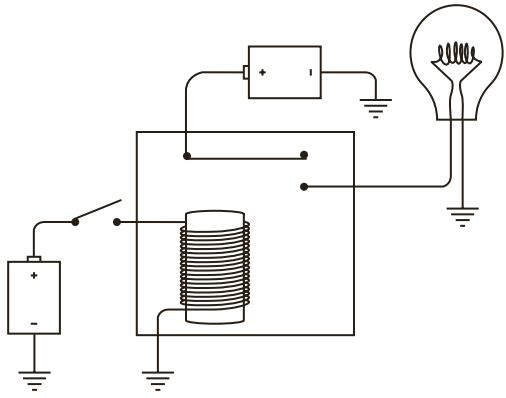
Обратите внимание: переключатель слева разомкнут, а лампочка не горит. Когда вы замкнете переключатель, ток из батарейки слева от него потечет по виткам катушки, намотанной на железный сердечник, который приобретет магнитные свойства и притянет гибкую металлическую полоску, что, в свою очередь, приведет к замыканию цепи и включению лампочки.



Когда электромагнит притягивает металлическую полоску, реле считается *активированным*. После размыкания выключателя железный сердечник теряет магнитные свойства, а металлическая полоска возвращается в исходное положение.

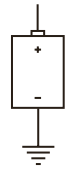
Такой способ зажечь лампочку кажется довольно мудреным, и это действительно так. Если бы мы хотели ограничиться только включением лампочки, мы могли бы обойтись и без реле. Однако перед нами стоит более сложная задача.

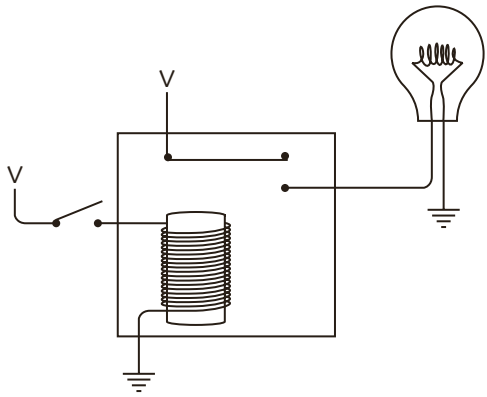
В этой главе мы будем часто использовать реле (а после сборки логических вентилей полностью от них откажемся), поэтому хочу упростить схему. Мы можем избавиться от некоторых проводов с помощью земли. В данном примере «земля» — просто общий провод; к реальной земле ничего подключать не нужно.



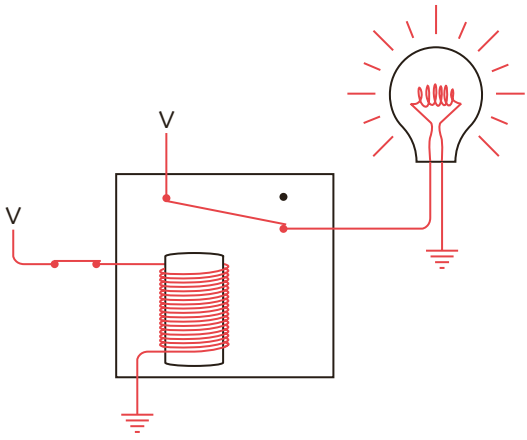
Понимаю, это не похоже на упрощение, однако мы еще не закончили. Важно: отрицательные контакты обеих батарей подключены к земле.

Так что везде, где нам встретится подобное изображение, заменим его заглавной буквой *V* (которая означает voltage — «напряжение»), как сделали это в главах 5 и 6. Теперь реле выглядит так.

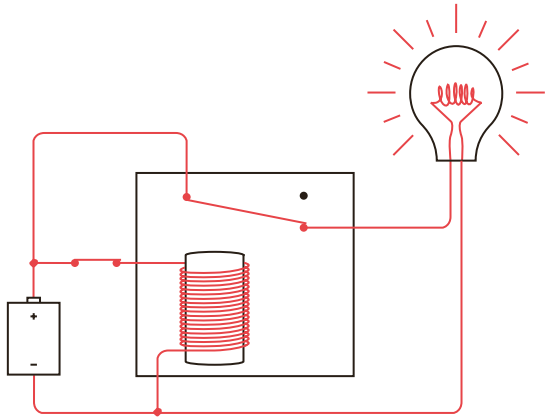




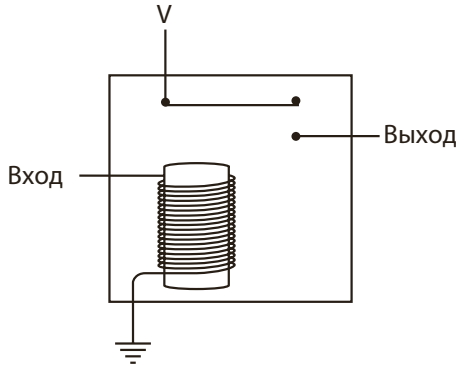
Когда переключатель замкнут, ток между источником питания (V) и землей течет через катушку электромагнита. Это заставляет электромагнит притянуть гибкую металлическую полоску, которая замыкает цепь между источником питания, лампочкой и землей, и лампочка загорается.



На этих схемах присутствуют два источника питания и две земли, однако все источники питания, как и все земли, на приведенных в этой главе схемах могут быть соединены друг с другом. Все схемы, состоящие из реле и логических вентилей, изображенные в этой и следующей главах, допускают использование только одной (хотя и мощной) батарейки. Например, предыдущую схему можно перерисовать только с одним источником питания.

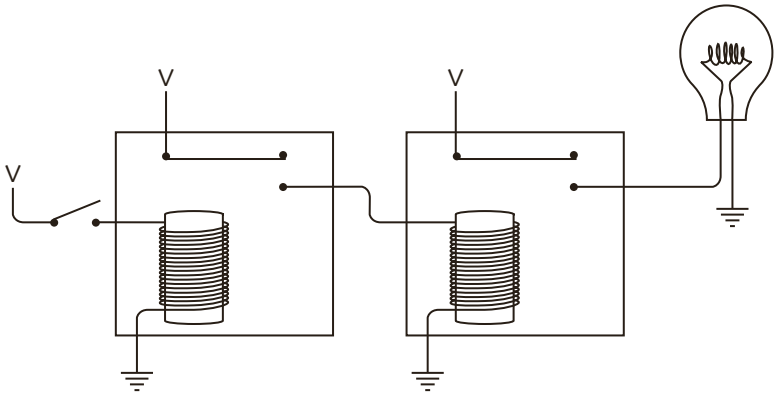


Учитывая то, что мы собираемся делать с реле, эта схема не является достаточно понятной. Лучше избегать замкнутых цепей и рассматривать работу реле, как и в случае с описанным ранее пультом управления, с точки зрения *входного* и *выходного* сигналов.

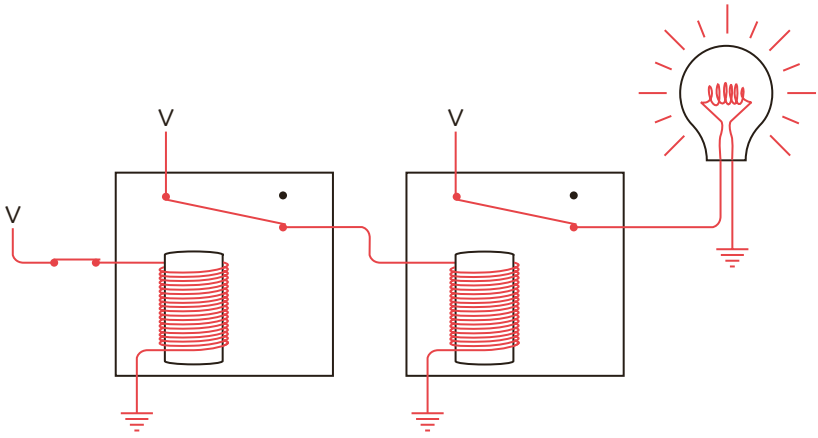


Если напряжение поступает на вход (например, если он соединен с источником питания с помощью переключателя), то активируется электромагнит, и на выходе появляется напряжение.

Ко входу реле не обязательно подключать переключатель, а к выходу — лампочку. Выход одного реле может быть подключен ко входу другого.

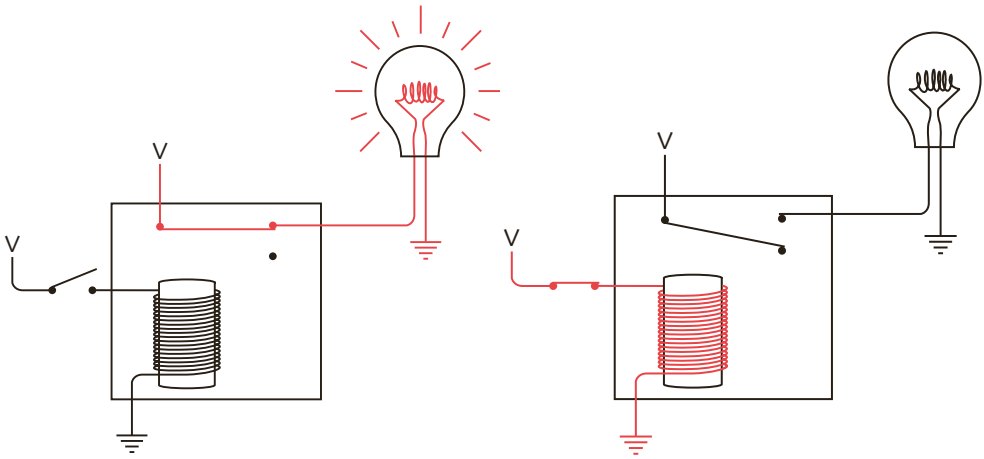


Замыкание переключателя активирует первое реле, которое затем подает напряжение на второе. Срабатывание второго реле приведет к включению лампочки.



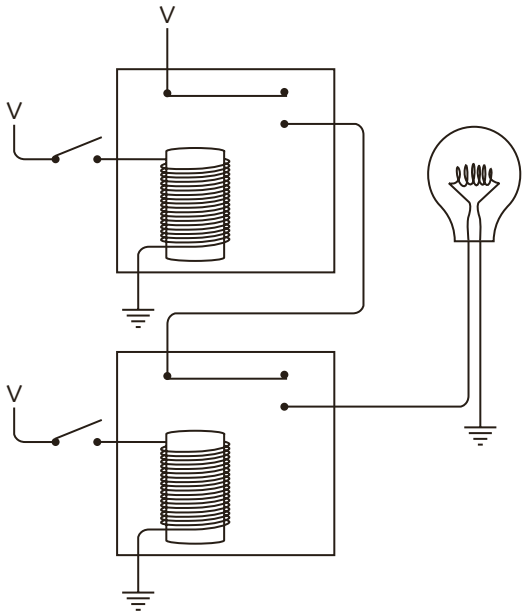
Соединяя несколько реле, можно конструировать логические вентили.

На самом деле лампочку можно подключить к реле двумя способами. Обратите внимание на гибкую металлическую деталь, которую притягивает электромагнит. В состоянии покоя она касается одного контакта. Когда электромагнит притягивает ее, она касается другого контакта. Мы использовали нижний контакт в качестве выхода реле, однако могли бы применить и верхний. В этом случае выход реле меняется на противоположный, и лампочка загорается при размыкании входного переключателя. При замыкании входного переключателя лампочка гаснет.

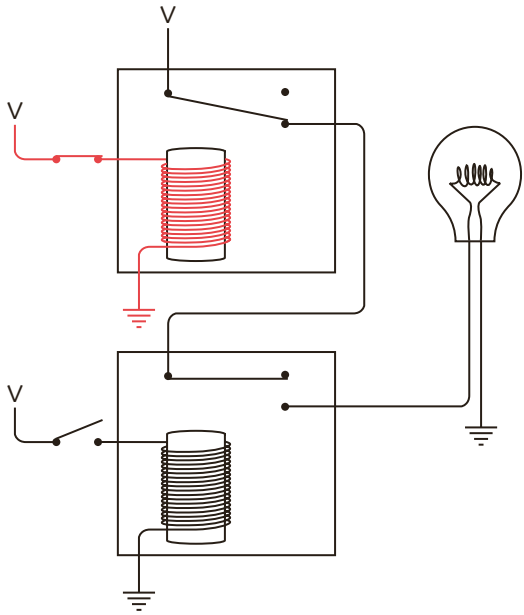


Реле такого типа называется *двухпозиционным*. Оно имеет два электрически противоположных выхода. Когда на одном из них есть напряжение, на другом его нет.

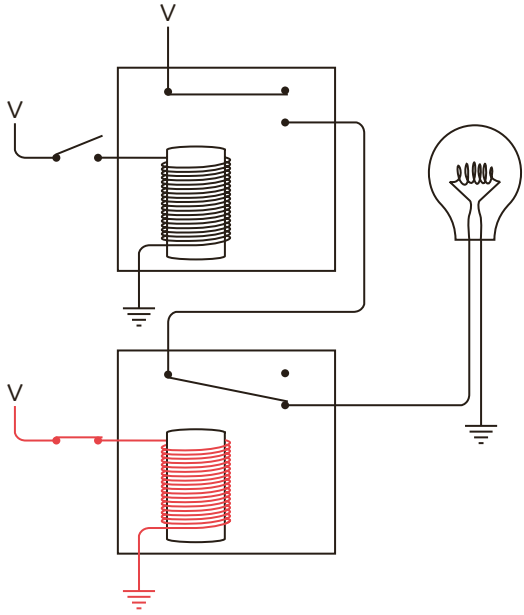
Как и в случае с переключателями, два реле могут быть соединены последовательно.



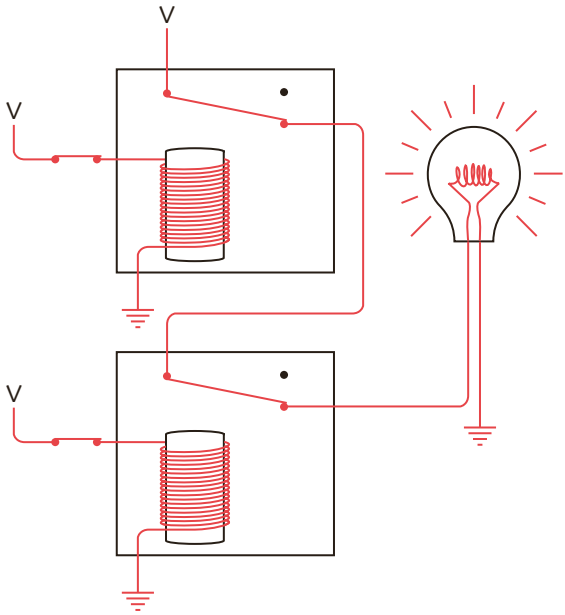
Выход верхнего реле подает напряжение на нижнее. Как видите, когда оба переключателя разомкнуты, лампочка не горит. Попробуем замкнуть верхний переключатель.



Лампочка не загорается, поскольку нижний переключатель все еще разомкнут, и второе реле не срабатывает. Попробуем разомкнуть верхний переключатель и замкнуть нижний.

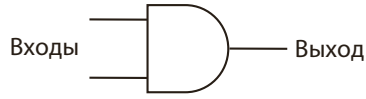


Лампочка по-прежнему не горит. Ток до нее не доходит, потому что не сработало первое реле. Единственным способом зажечь лампочку является замыкание обоих переключателей.



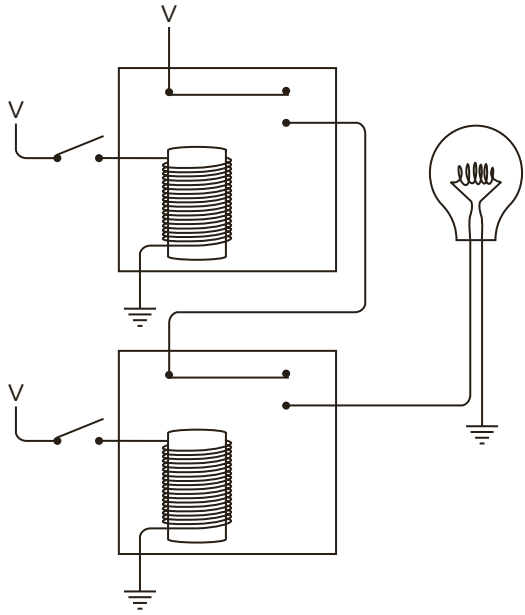
Теперь активированы оба реле, и между источником питания, лампочкой и землей течет ток.

Подобно двум соединенным последовательно переключателям, эти два реле решают небольшую логическую задачу. Лампа загорается только в случае срабатывания обоих реле. Такая схема последовательного соединения двух реле называется *вентилем И*. Для его обозначения на схемах инженеры-электрики используют специальный символ, который выглядит так.

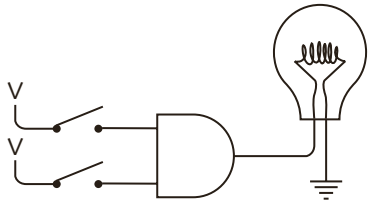


Это первый из четырех основных логических вентилей. Вентиль И имеет два входа (слева на приведенной выше схеме) и один выход (справа). Вам часто будет встречаться именно такое изображение вентиля И — со входами слева и выходом справа. Дело в том, что людям, привыкшим читать слева направо, удобнее изучать электрические схемы также слева направо. Однако входы логического вентиля И можно было бы изобразить вверху, справа или внизу.

Исходная схема с двумя последовательно соединенными реле, двумя переключателями и лампочкой выглядела так.

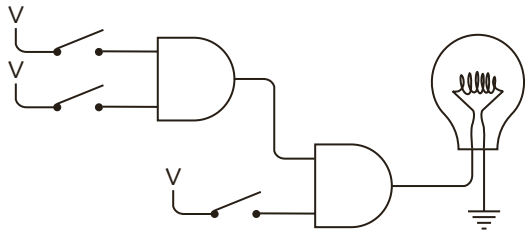


С использованием символа вентиля И эта же схема принимает следующий вид.



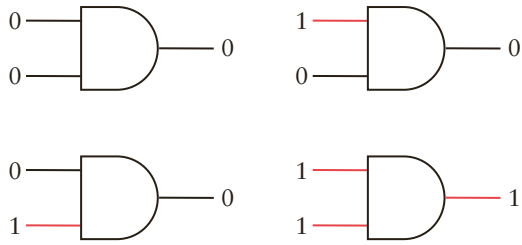
Обратите внимание: символ вентиля И не только используется вместо двух соединенных последовательно реле, но также подразумевает, что верхнее реле подключено к источнику питания и оба реле соединены с землей. Опять же, лампочка загорается только в случае замыкания верхнего *и* нижнего переключателей. Вот почему эта схема называется *вентилем И*.

Входы вентиля И не обязательно должны быть соединены с переключателями, а выход — подключен к лампочке. В данном случае мы имеем дело просто с напряжением на входах и выходе. Например, выход одного вентиля И может быть входом второго вентиля И.



Эта лампочка загорится лишь в случае замыкания всех трех переключателей. Только если будут замкнуты верхние два переключателя, выход первого вентиля И активирует первое реле во втором вентиле И. Нижний переключатель активирует второе реле во втором вентиле И.

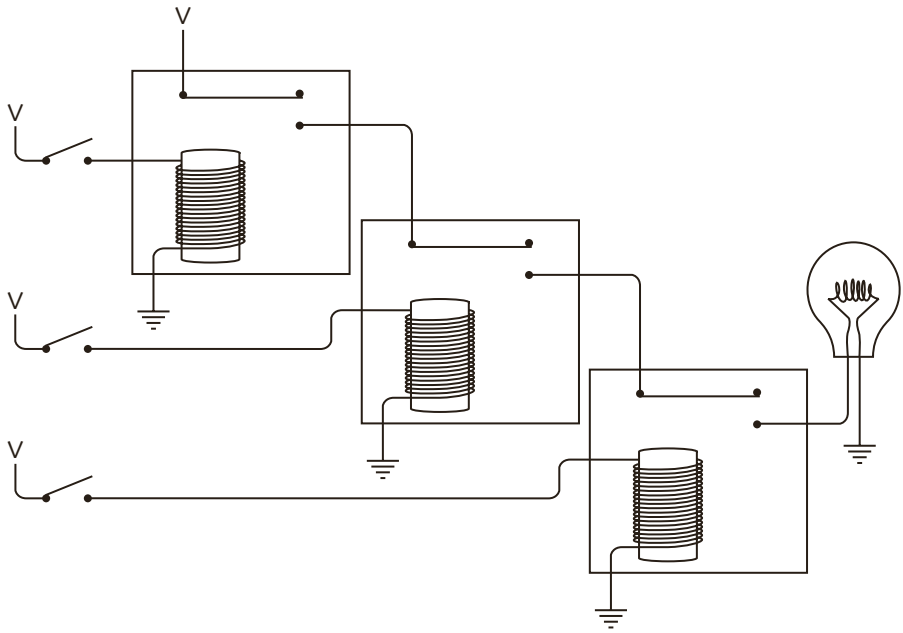
Если мы выразим отсутствие напряжения в виде 0, а его наличие — в виде 1, то зависимость выходного сигнала вентиля И от входных сигналов будет следующей.



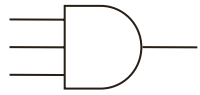
Как и в случае с двумя последовательно соединенными переключателями, работу вентиля И можно описать с помощью небольшой таблицы.

И	**0**	**1**
0	0	0
1	0	1

Можно делать вентиль И с более чем двумя входами. Например, вы последовательно соединили три реле.

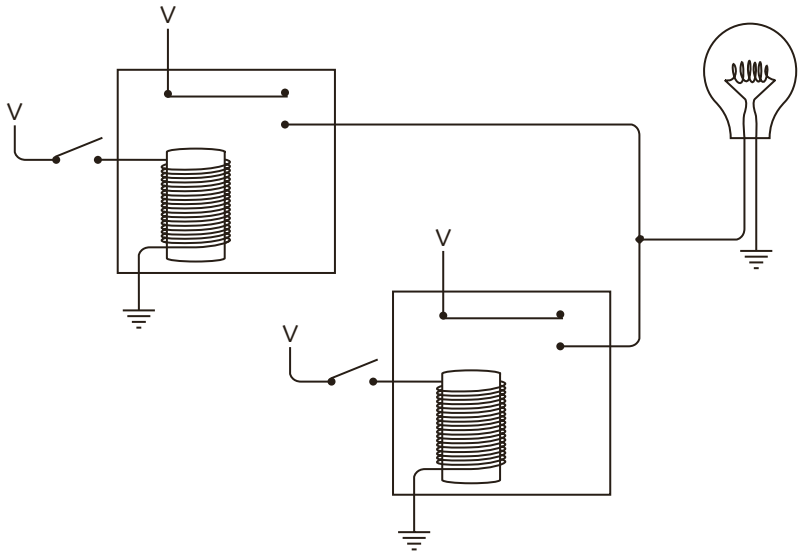


Лампочка загорится при замыкании всех трех переключателей. Подобная конфигурация обозначается таким символом.

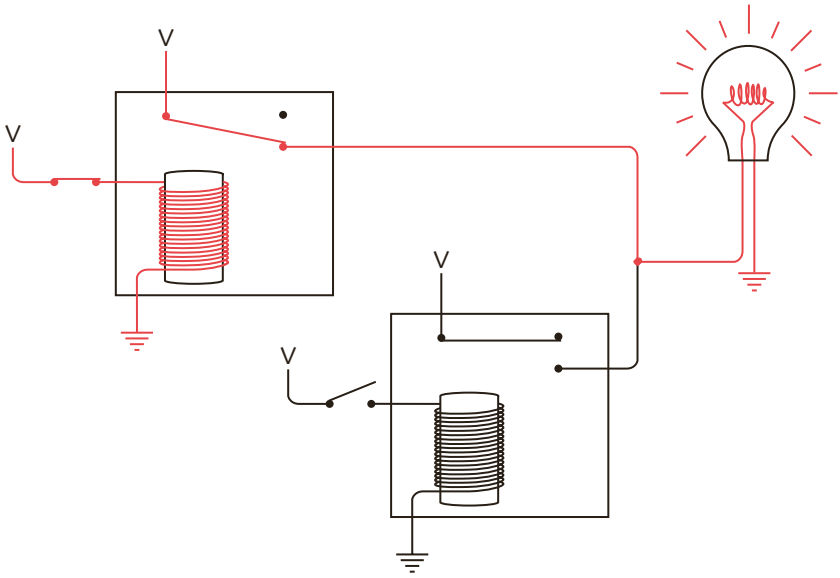


Такая схема называется *трехвходовым вентилем И*.

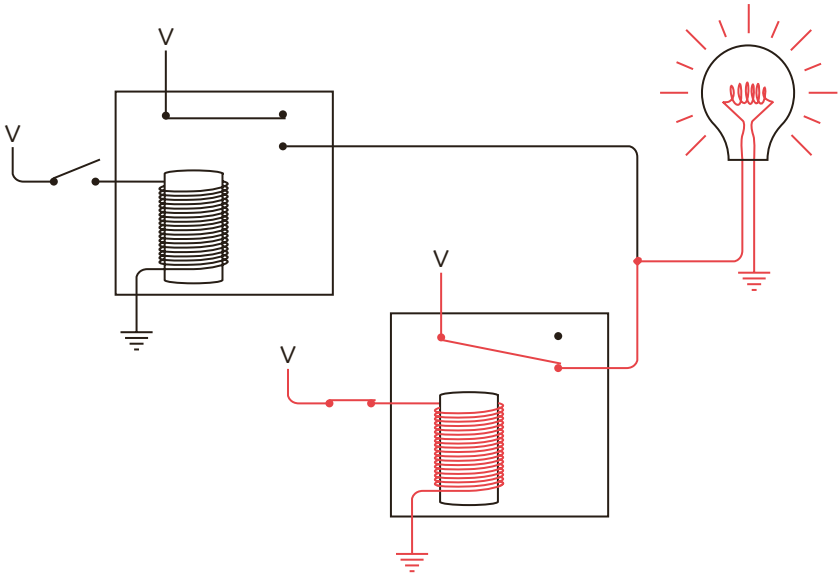
Следующий логический вентиль состоит из двух реле, соединенных параллельно.



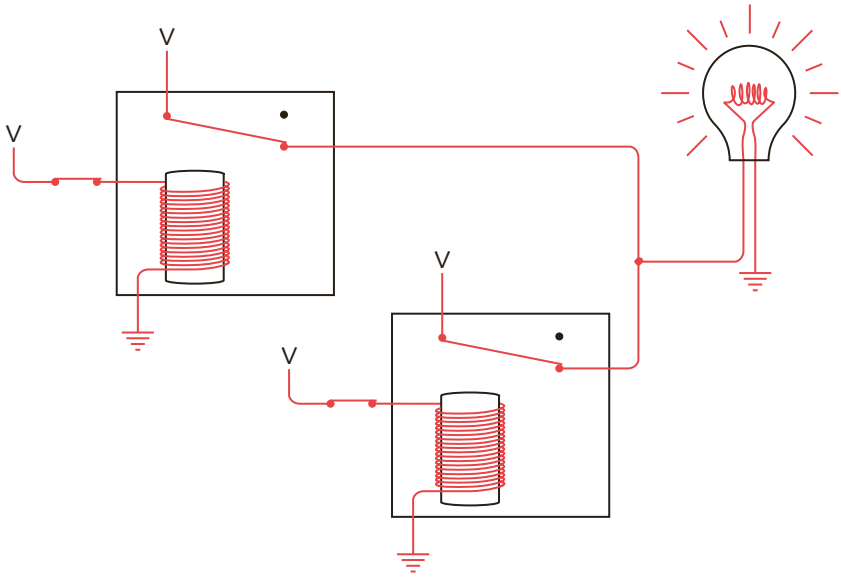
Важно: выходы двух реле соединены друг с другом. Этот объединенный выход подает питание на лампочку. Для того чтобы лампочка загорелась, достаточно активировать одно из двух реле. Например, если мы замкнем верхний переключатель, лампочка загорится, поскольку получает питание от левого реле.



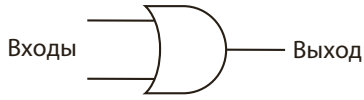
Аналогично лампочка загорится, если мы оставим верхний выключатель разомкнутым, но замкнем нижний.



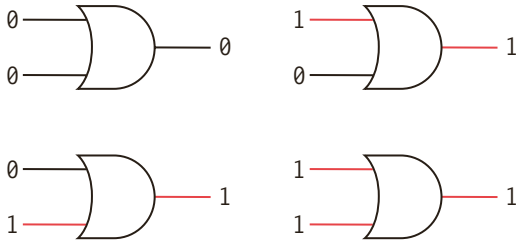
Лампочка также загорится при замыкании обоих переключателей.



В данном случае мы находимся в ситуации, когда лампочка загорается при замыкании верхнего *или* нижнего переключателя. Ключевым здесь является слово *или*, поэтому данная схема называется *вентилем ИЛИ*. Для его обозначения инженеры-электрики используют такой символ.



Он несколько похож на символ вентиля И, за исключением закругления стороны входов. На выходе вентиля ИЛИ есть напряжение, если оно подается на один из двух его входов. Если мы обозначим отсутствие напряжения 0, а его наличие — 1, то вентиль ИЛИ сможет находиться в четырех возможных состояниях.

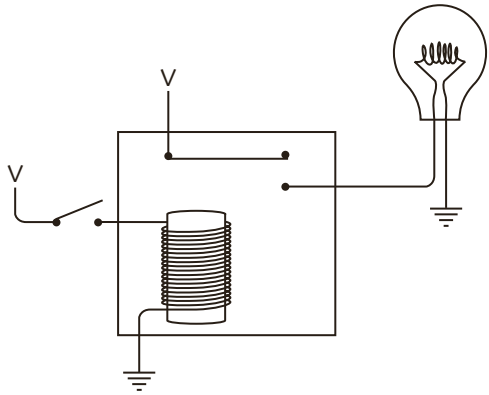


Результаты работы вентиля ИЛИ можно представить в виде таблицы.

ИЛИ	0	1
0	0	1
1	1	1

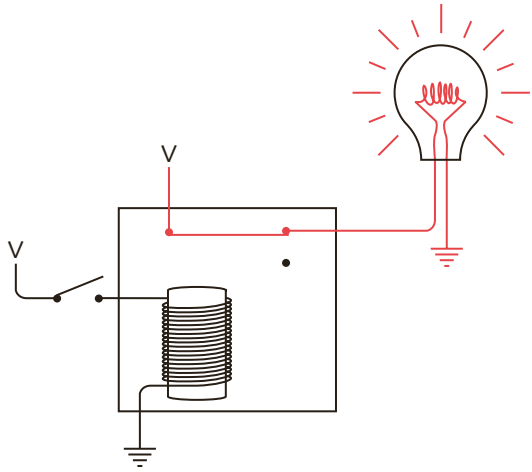
Вентиль ИЛИ также может иметь более двух входов. Выход такого вентиля равен 1, если любой из его входов равен 1; выход вентиля равен 0, если все его входы равны 0.

Ранее я объяснил, что используемые нами реле называются двухпозиционными, потому что их выходы могут быть подключены двумя разными способами. Как правило, при разомкнутом переключателе лампочка не горит.

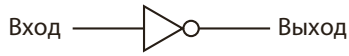


При замыкании переключателя лампочка загорается.

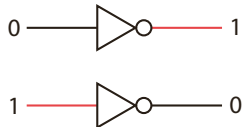
Кроме того, вы можете использовать другой контакт, чтобы лампочка загоралась при *размыкании* переключателя.



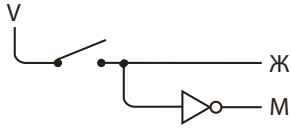
В этом случае лампочка будет гаснуть при замыкании переключателя. Подключенное таким способом одиночное реле называется *инвертором*. Инвертор не является логическим вентилем (логические вентили всегда имеют два или более входов), однако он часто оказывается очень полезным и изображается с помощью специального символа.



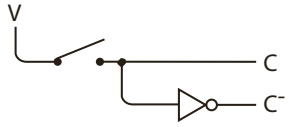
Данная схема называется инвертором, потому что она инвертирует 0 (отсутствие напряжения) в 1 (наличие напряжения) и наоборот.



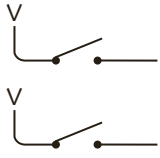
Теперь, когда у нас есть инвертор, вентиль *И* и вентиль *ИЛИ*, мы можем приступить к созданию пульта управления, который позволит автоматизировать выбор идеальной кошки. Начнем с переключателей. Первый переключатель в замкнутом состоянии соответствует кошке, в разомкнутом — коту. Так мы сможем генерировать два сигнала, которые обозначим Ж и М.



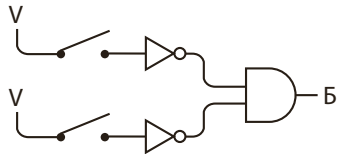
Ж равно 1, М равно 0, и наоборот. Аналогично второй переключатель соответствует стерилизованной кошке в замкнутом состоянии, нестерилизованной — в разомкнутом.



Со следующими двумя переключателями дело обстоит чуть сложнее. Различные комбинации должны соответствовать четырем разным цветам. Вот два переключателя, подключенных к источнику питания.

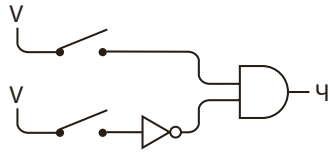


Когда оба переключателя разомкнуты (как показано на рисунке), они соответствуют белому цвету. Вот как можно использовать два инвертора и один вентиль И для того, чтобы сгенерировать сигнал, обозначенный буквой *Б*, который будет равен 1 (наличие напряжения), если вы выбрали белую кошку, и 0 (отсутствие напряжения), если не выбрали.



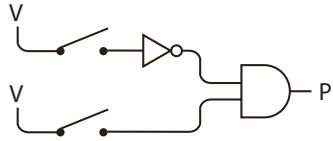
Когда переключатели разомкнуты, входы обоих инверторов равны 0. Таким образом, оба выходных сигнала инверторов (которые подаются на входы вентиля И) равны 1. Это означает, что выход вентиля И равен 1. При замыкании любого из переключателей выход вентиля И будет равен 0.

Чтобы выбрать черную кошку, мы замыкаем первый переключатель. Это можно сделать, используя один инвертор и вентиль И.

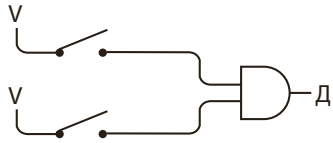


Выход вентиля И будет равен 1 только в том случае, если первый переключатель замкнут, а второй — разомкнут.

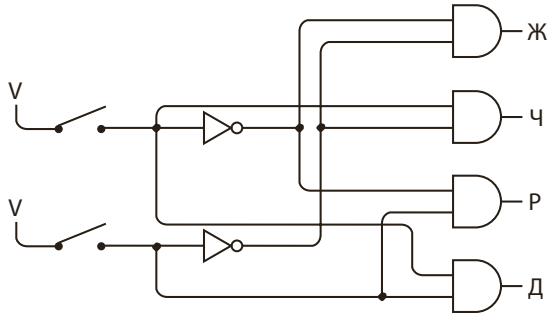
Аналогично замыкание второго переключателя будет означать выбор рыжей кошки.



Замыкание обоих переключателей означает, что нам нужна кошка «другого» цвета.



Теперь давайте объединим описанные выше четыре небольшие схемы в одну. (Как обычно, черными точками обозначаются соединения проводов; провода, на пересечениях которых черных точек нет, *не соединены.*)



Понимаю, в этих хитросплетениях проводов сложно разобраться. Однако если вы внимательно проследите, откуда подаются сигналы на входы каждого из вентилей И, и проигнорируете то, куда еще они идут, то увидите, что схема работает. Если оба переключателя разомкнуты, выход Б будет равен 1, а остальные — 0. Если первый переключатель замкнут, выход Ч будет равен 1, а остальные — 0 и т. д.

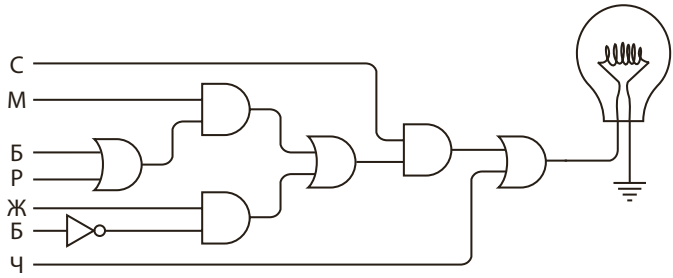
Для соединения вентилей и инверторов существует несколько простых правил: выход одного вентиля (или инвертора) может являться входом одного или нескольких других вентилей (инверторов). Однако выходы двух или более вентилей (инверторов) никогда не соединяются друг с другом.

Эта схема, состоящая из четырех вентилей И и двух инверторов, называется *дешифратором двух линий на четыре*. На его вход подается два бита, которые в различных комбинациях могут представлять четыре разных значения. На его выходе образуются четыре сигнала, лишь один из которых равен 1 в любой момент времени (какой конкретно, зависит от входных значений). По аналогичному принципу вы можете создать дешифратор трех линий на восемь или дешифратор четырех линий на шестнадцать и т. д.

Еще раз приведу упрощенное логическое выражение для выбора кошки:

$$(С \times ((М \times (Б + Р)) + (Ж \times (1 - Б)))) + Ч.$$

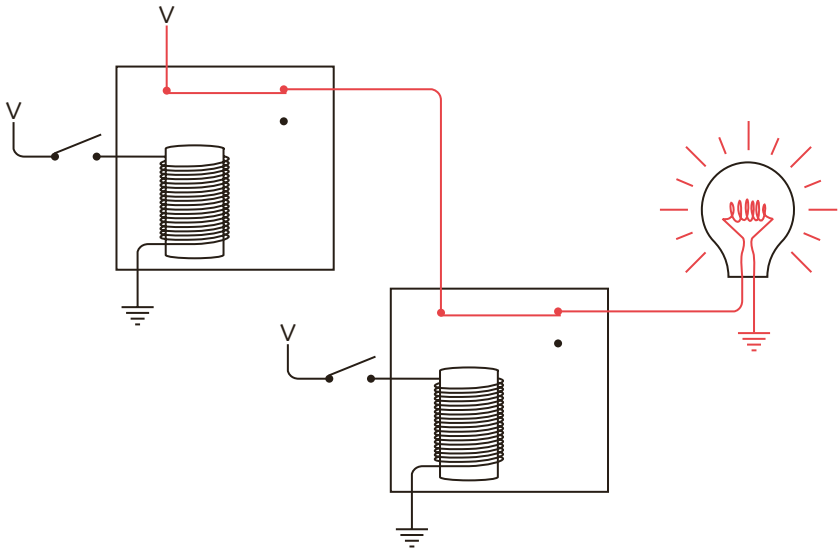
Каждому знаку «+» в этом выражении должен соответствовать вентиль ИЛИ, а каждому знаку «×» — вентиль И.



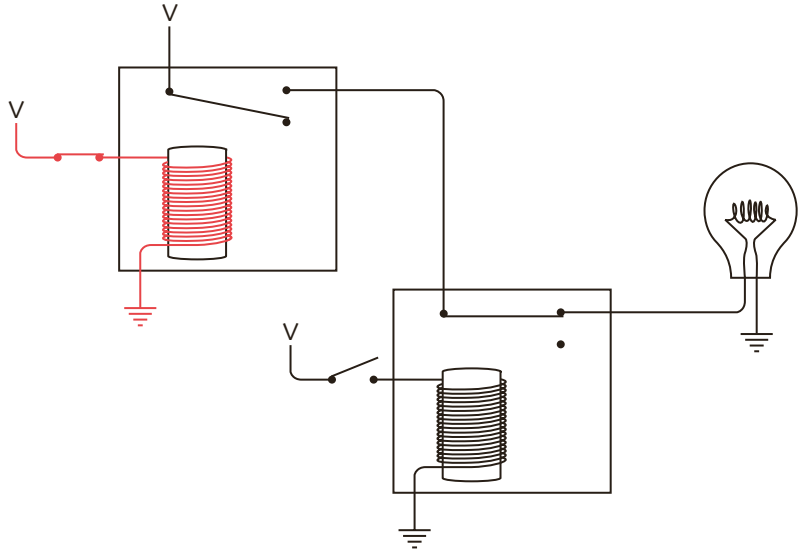
Порядок символов слева от схемы соответствует их порядку в выражении. Эти сигналы поступают от переключателей, соединенных с инверторами и дешифратором «2 на 4». Обратите внимание на использование инвертора для реализации части выражения 1 – Б.

Вы можете подумать, что в схеме используется слишком много реле, и это действительно так. В ней присутствуют два реле для каждого вентиля И и ИЛИ и одно реле для каждого инвертора. Могу лишь посоветовать привыкнуть к такому положению дел. В следующих главах мы будем использовать гораздо больше реле. Просто радуйтесь, что вам не придется покупать реле и собирать эти схемы дома.

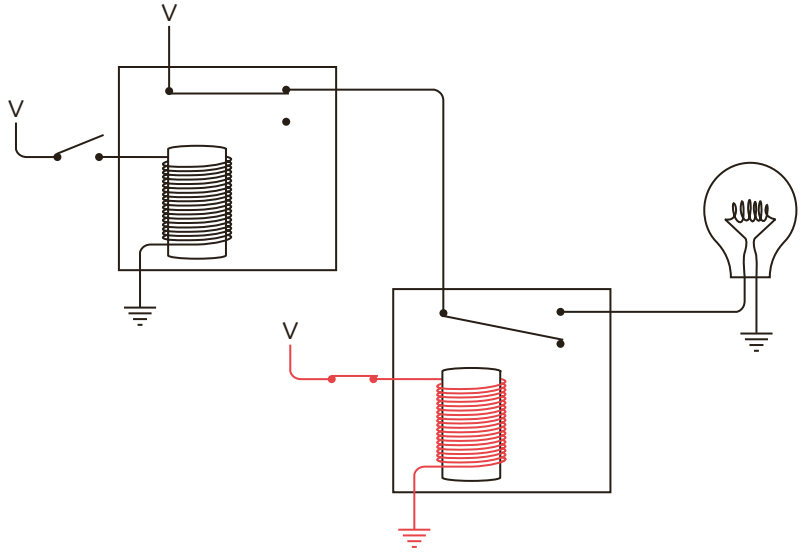
Сейчас мы рассмотрим еще два логических вентиля. Оба используют выход реле, на котором присутствует напряжение, когда реле не активировано (выход, используемый в инверторе). Например, в следующей конфигурации выход одного реле подает питание на вход второго. Когда оба входа отключены от источника питания, лампочка горит.



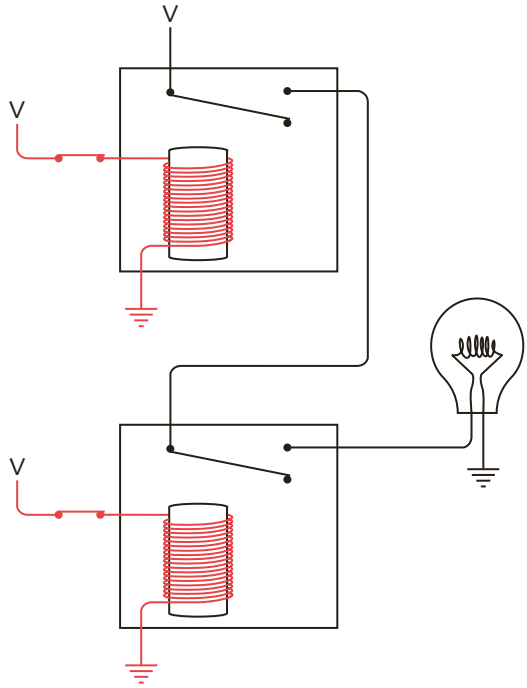
При замыкании верхнего выключателя лампочка гаснет.



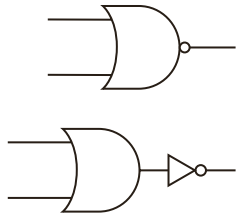
Лампочка гаснет из-за того, что на второе реле не подается питание. Точно так же лампочка гаснет при замыкании нижнего выключателя.



Когда замкнуты оба переключателя, лампочка тоже не горит.



Это поведение прямо противоположно поведению вентиля ИЛИ. Такая схема называется *вентилем ИЛИ-НЕ*.



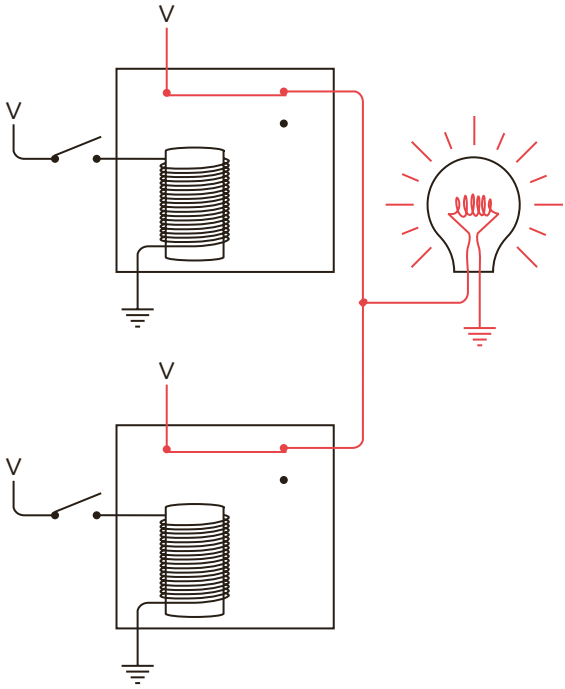
Его символ аналогичен символу вентиля ИЛИ, за исключением того, что на выходе изображен небольшой кружок, означающий *инвертировать*. Вентиль ИЛИ-НЕ соответствует следующей схеме.

Результаты работы вентиля ИЛИ-НЕ представлены в таблице.

ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

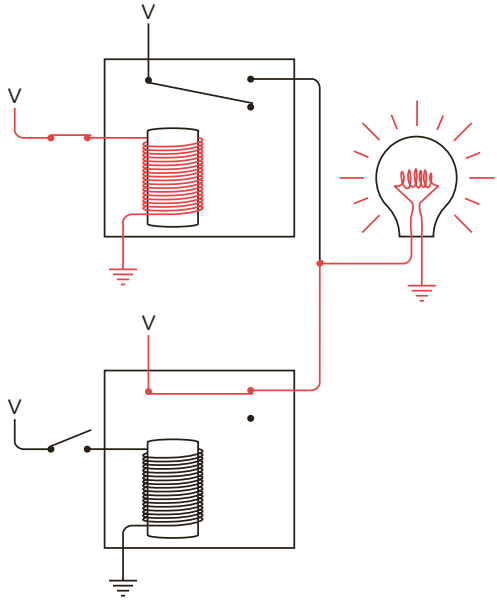
Они противоположны результатам работы вентиля ИЛИ, выход которого равен 1, если один из двух его входов равен 1, а 0 — только если оба входа равны 0.

Далее показан еще один способ соединения двух реле.

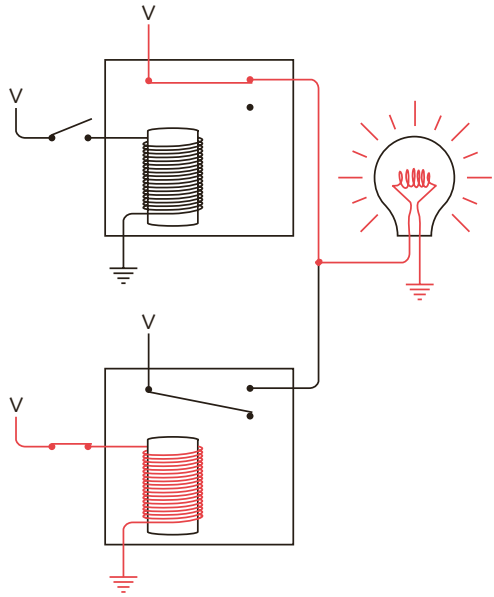


В данном случае два выхода соединены. Это похоже на конфигурацию вентиля ИЛИ, только здесь используются другие контакты. Лампочка горит, когда оба переключателя разомкнуты.

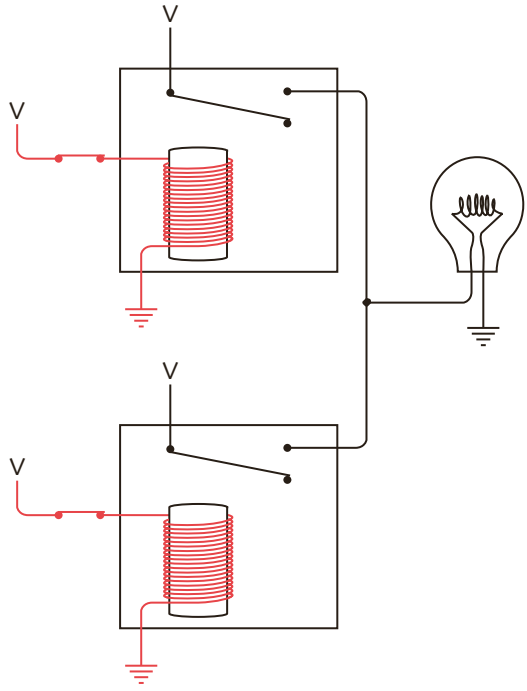
Лампочка продолжает гореть при замыкании верхнего переключателя.



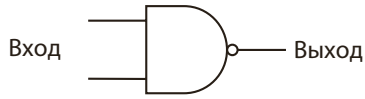
Точно так же лампочка продолжает гореть и при замыкании нижнего переключателя.



Только при замыкании обоих переключателей лампочка гаснет.



Это поведение прямо противоположно поведению вентиля И. Такая схема называется *вентилем И-НЕ*. Вентиль И-НЕ изображается так же, как и вентиль И, но с кружком на выходе, который означает, что выходной сигнал противоположен выходному сигналу вентиля И.



Вентиль И-НЕ демонстрирует следующее поведение.

И-НЕ	**0**	**1**
0	1	1
1	1	0

Обратите внимание: выход вентиля И-НЕ противоположен выходу вентиля И. Выход вентиля И равен 1, только если оба входа равны 1; в противном случае выход равен 0.

Итак, мы рассмотрели четыре различных способа подключения реле, которые имеют два входа и один выход. Каждая конфигурация дает несколько различающиеся результаты. Для экономии сил и времени мы назвали эти конфигурации логическими вентилями и решили обозначать их с помощью символов, используемых инженерами-электриками. Выходной сигнал конкретного логического вентиля зависит от входного сигнала, как показано в следующих таблицах.

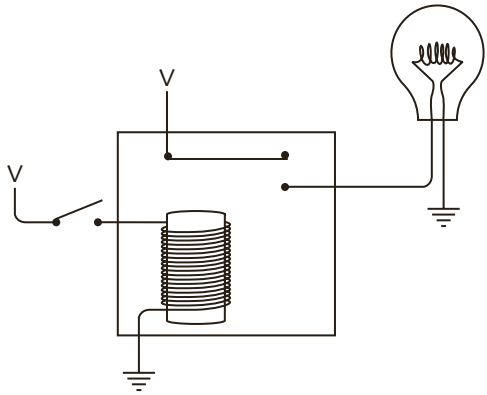
И	**0**	**1**
0	0	0
1	0	1

ИЛИ	**0**	**1**
0	0	1
1	1	1

И-НЕ	**0**	**1**
0	1	1
1	1	0

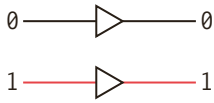
ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

Теперь у нас есть четыре логических вентиля и инвертор. Осталось дополнить инструментарий обычным реле, которое называется *буфером*.



Буфер изображается так.

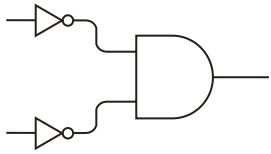
Этот символ аналогичен символу инвертора, но без маленького кружка. Буфер примечателен тем, что он почти ничего не делает. Выходной сигнал буфера совпадает с его входным сигналом.



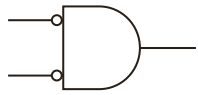
Однако вы можете использовать буфер при наличии слабого входного сигнала. Как вы помните, именно по этой причине реле использовались в телеграфной системе много лет назад. Кроме того, буфер можно применять для небольшой задержки сигнала. Дело в том, что для срабатывания реле требуется немного времени — небольшая доля секунды.

Отныне в книге редко будут встречаться изображения реле. Вместо этого следующие схемы будут состоять из буферов, инверторов, четырех основных логических вентилей и более сложных схем (дешифратора «2 на 4», например), собранных из этих вентилей. Разумеется, все эти компоненты состоят из реле, однако нам нет необходимости их рассматривать.

Ранее, когда мы конструировали дешифратор «2 на 4», нам встретилась небольшая схема следующего типа.

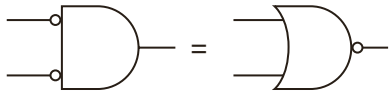


Два инвертированных входа стали входами вентиля И. Иногда такая конфигурация изображается без инверторов.



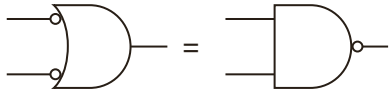
Обратите внимание на маленькие кружки на входе вентиля И, которые указывают, что сигналы в этой точке инвертируются: 0 (отсутствие напряжения) становится 1 (наличие напряжения), и наоборот.

Вентиль И с двумя инвертированными входами ведет себя точно так же, как вентиль ИЛИ-НЕ.



Выход равен 1, только если оба входа равны 0.

Аналогично вентиль ИЛИ с двумя инвертированными входами эквивалентен вентилю И-НЕ.



Выход равен 0, только если оба входа равны 1.

Эти две пары эквивалентных схем представляют электрическое воплощение законов Огастеса де Моргана, еще одного математика викторианской эпохи, который был на девять лет старше Буля. Его книга «Формальная логика» была опубликована в 1847 году, согласно преданию, в один день с книгой Буля «Математический анализ логики». На самом деле на занятия логикой Буля подвигла открытая вражда между де Морганом и другим британским математиком, связанная с обвинениями в плагиате (история оправдала де Моргана). С самого начала де Морган осознал важность прозрений Буля. Он бескорыстно поощрял Буля и помогал ему в исследованиях, однако сегодня он, к сожалению, почти забыт, а в памяти потомков остались только его знаменитые законы.

Законы де Моргана проще всего выразить следующим образом.

$$\overline{A} \times \overline{B} = \overline{A + B}$$
$$\overline{A} + \overline{B} = \overline{A \times B}$$

A и B — два булевых операнда. В первом выражении они инвертируются, а затем объединяются с помощью булева оператора И. Это эквивалентно объединению двух операндов с помощью булева оператора ИЛИ и последующему инвертированию результата (соответствует оператору ИЛИ-НЕ). Во втором выражении два оператора инвертируются, а затем объединяются с помощью булева оператора ИЛИ. Это эквивалентно объединению двух операндов с помощью булева оператора И и последующему инвертированию результата (соответствует оператору И-НЕ).

Законы де Моргана — важный инструмент для упрощения булевых выражений, а значит, для упрощения схем. История показала, что именно в этом заключалось значение работы Клода Шеннона для инженеров-электриков. Однако чрезмерное упрощение схем не является целью этой книги. Нам важно собрать работающую схему, а не сделать так, чтобы она работала как можно проще. Этим мы и займемся в следующей главе — соберем работающую счетную машину.

Глава 12
Двоичный сумматор

Сложение — простейшая арифметическая операция. Если мы хотим создать компьютер (а именно в этом заключается цель этой книги), сначала нужно найти способ создания устройства, складывающего два числа. По сути, компьютеры выполняют только операцию сложения. Если нам удастся сконструировать механизм, умеющий складывать, мы окажемся способны создать устройство, использующее операцию сложения для того, чтобы вычитать, умножать, делить, рассчитывать платежи по ипотеке, отправлять ракеты на Марс, играть в шахматы и вносить путаницу в наши телефонные счета.

Сумматор, который мы построим, будет большим, нескладным, медленным и шумным по сравнению с современными калькуляторами и компьютерами. Самое интересное заключается в том, что мы соберем эту машину из простых электрических устройств, о которых говорили в предыдущих главах, — переключателей, лампочек, проводов, батарейки и реле, объединенных в различные логические вентили. Этот сумматор будет состоять исключительно из деталей, которые уже были изобретены 120 лет назад. Особенно хорошо то, что нам не нужно ничего собирать в своей гостиной; вместо этого мы можем конструировать на бумаге и в уме.

Эта машина будет работать исключительно с двоичными числами, в ней будут отсутствовать некоторые современные функции. Вы не сможете использовать клавиатуру для ввода чисел, подлежащих сложению; вместо этого будет ряд переключателей. Роль дисплея для отображения результатов в этом сумматоре исполнит ряд лампочек.

Однако машина сумеет сложить два числа, и она сделает это фактически как компьютер.

Сложение двоичных чисел похоже на сложение десятичных. Если хотите сложить два десятичных числа, например 245 и 673, вы разбиваете задачу на более простые этапы. На каждом этапе складываете две десятичные цифры. В данном примере начинаете со сложения 5 и 3. Эта задача решается быстрее, если вы знаете таблицу сложения.

Большая разница между сложением десятичных и двоичных чисел заключается в том, что в случае с двоичными числами используется более простая таблица.

+	**0**	**1**
0	0	1
1	1	10

Если вы выросли среди дельфинов, вероятно, вы в школе учили эту таблицу, громко произнося:

0 плюс 0 равно 0,
0 плюс 1 равно 1,
1 плюс 0 равно 1,
1 плюс 1 равно 0, 1 в уме.

Вы можете добавить в эту таблицу нули так, чтобы каждый результат представлял 2-битное значение.

+	**0**	**1**
0	00	01
1	01	10

Таким образом, результатом сложения пары двоичных чисел являются два бита, которые называются *разрядом суммы* и *разрядом переноса* (1 плюс 1 равно 0, 1 в уме). Теперь мы можем разделить таблицу сложения двоичных чисел на две таблицы. Первая — для разряда суммы.

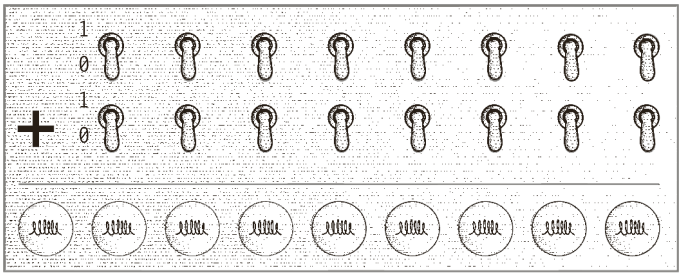
+ **сумма**	**0**	**1**
0	0	1
1	1	0

Вторая — для разряда переноса.

+ **перенос**	**0**	**1**
0	0	0
1	0	1

Сложение двоичных чисел удобно рассматривать так, поскольку наш сумматор выполняет операции суммирования и переноса отдельно. Для создания двоичного сумматора потребуется сконструировать схему, выполняющую эти операции. Работа исключительно в двоичной системе счисления значительно упрощает задачу, поскольку все части схемы — переключатели, лампочки и провода — могут представлять двоичные цифры.

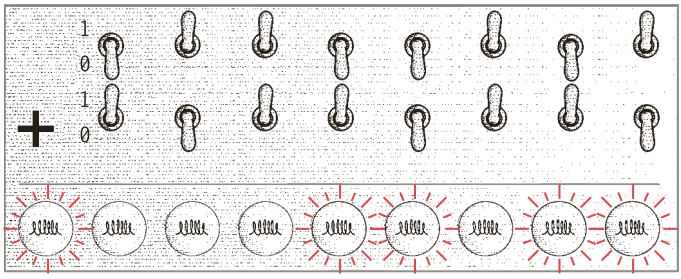
Как и при сложении десятичных чисел, мы складываем двоичные числа столбец за столбцом, начиная с крайнего правого.



Обратите внимание: при сложении значений в третьем столбце справа 1 переносится в следующий столбец. Это происходит снова в шестом, седьмом и восьмом столбцах справа.

Какого размера двоичные числа мы хотим сложить? Поскольку мы создаем сумматор прямо в уме, то можем сделать так, чтобы он складывал очень длинные числа. Однако давайте будем благоразумными и ограничимся двоичными числами длиной до восьми бит, то есть будем складывать двоичные числа в диапазоне от 0000 0000 до 1111 1111 (десятичные от 0 до 255). Сумма двух 8-битных чисел может достигать двоичного значения 1 1111 1110 (десятичного значения 510).

Пульт управления нашим двоичным сумматором может выглядеть так.



На этом пульте есть два ряда по восемь переключателей. Этот набор переключателей — устройство ввода, которое мы будем использовать для ввода двух 8-битных значений. В этом устройстве выключенный переключатель (положение

вниз) соответствует значению 0, а включенный (положение вверх) — 1, как в случае с настенными переключателями в вашем доме. Устройство вывода в нижней части пульта — ряд из девяти лампочек, которые отобразят результат сложения. Негорящая лампочка соответствует значению 0, горящая — 1. Нам требуется девять лампочек, поскольку сумма двух 8-битных чисел может быть 9-битным числом.

В остальном сумматор будет состоять из логических вентилей, соединенных различными способами. Переключатели будут активировать реле в логических вентилях, которые, в свою очередь, будут зажигать нужные лампочки. Например, если мы хотим сложить числа 01100101 и 10110110 (из предыдущего примера), включаем соответствующие переключатели.

Загоревшиеся лампочки показывают результат: 100011011. По крайней мере, мы на это надеемся. Мы ведь еще не собрали устройство!

В предыдущей главе я упомянул, что в этой книге буду использовать множество реле. Для 8-битного сумматора, который мы создаем, требуется не менее 144 реле — по восемнадцать для каждой из восьми пар битов, которые складываем. Если бы я показал готовую схему, вы бы наверняка испугались. Никому не под силу разобраться в схеме, состоящей из ста сорока четырех хитро соединенных реле. Вместо этого мы будем решать такую задачу поэтапно, используя логические вентили.

Возможно, вы сразу заметили связь между логическими вентилями и сложением двоичных чисел, когда увидели таблицу для разряда переноса, который возникает в результате сложения двух однобитных чисел.

+ перенос	**0**	**1**
0	0	0
1	0	1

Вероятно, вы узнали в ней результат работы вентиля И.

И	**0**	**1**
0	0	0
1	0	1

Таким образом, вентиль И вычисляет значение разряда переноса при сложении двух двоичных цифр.

Ага! Мы определенно делаем успехи. Наш следующий шаг, похоже, заключается в том, чтобы убедить некоторые реле вести себя так:

+ сумма	0	1
0	0	1
1	1	0

Это вторая часть задачи при сложении пары двоичных цифр. Вычислить значение разряда суммы оказывается не так просто, как значение разряда переноса, но мы справимся и с этой сложностью.

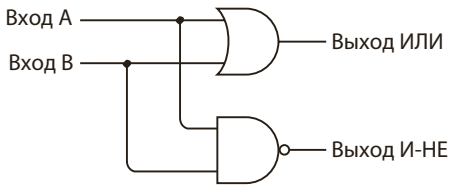
Первое, что нужно понять, — это то, что результат работы вентиля ИЛИ близок к тому, что нам нужно, за исключением значения в правом нижнем углу.

ИЛИ	0	1
0	0	1
1	1	1

Результат работы вентиля И-НЕ также близок к тому, что нам требуется, за исключением значения в верхнем левом углу:

И-НЕ	0	1
0	1	1
1	1	0

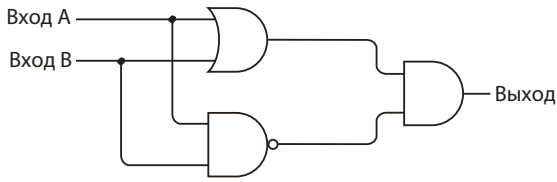
Итак, давайте подключим вентили ИЛИ и И-НЕ к одним и тем же входам.



В следующей таблице представлены выходные сигналы вентилей ИЛИ и И-НЕ и их сравнение с тем, что мы хотим получить от сумматора.

Вход A	Вход B	Выход ИЛИ	Выход И-НЕ	Что требуется
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Заметьте, что мы хотим получить значение 1, только если выходные сигналы обоих вентилей ИЛИ и И-НЕ равны 1. Это говорит о том, что эти два выходных сигнала могут являться входными сигналами для вентиля И.

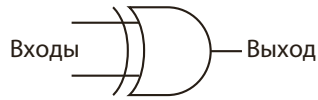


То, что нужно.

Обратите внимание: во всей этой схеме по-прежнему есть только два входа и один выход. Два входа относятся к обоим вентилям ИЛИ и И-НЕ. Выходные сигналы вентилей ИЛИ и И-НЕ подаются на вход вентиля И, и это дает именно тот результат, к которому мы стремимся.

Вход A	Вход B	Выход ИЛИ	Выход И-НЕ	Что требуется
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

На самом деле у этой схемы есть название: *вентиль исключающее ИЛИ* (Искл-ИЛИ, оно же — сложение по модулю 2). Она называется так потому, что выход равен 1, если вход A равен 1 *или* вход B равен 1, но не оба одновременно. Вместо того чтобы рисовать вентили ИЛИ, И-НЕ и И, мы можем использовать обозначение, которым инженеры-электрики показывают вентиль Искл-ИЛИ.



Это обозначение очень похоже на обозначение вентиля ИЛИ, но имеет дополнительную кривую линию со стороны входа.

Искл-ИЛИ	0	1
0	0	1
1	1	0

Вентиль Искл-ИЛИ — это последний логический элемент, который будет подробно описан в этой книге. Иногда в электротехнике используется шестой вентиль, называющийся *вентилем совпадения* или *эквивалентности*, поскольку выход равен 1 только при одинаковых сигналах на входе. Вентиль совпадения на выходе действует противоположно вентилю Искл-ИЛИ, поэтому его обозначение аналогично обозначению вентиля Искл-ИЛИ, но дополнено кружком со стороны выхода*.

Давайте повторим все, что уже знаем. При сложении двух двоичных чисел получается бит суммы и бит переноса.

+ сумма	0	1
0	0	1
1	1	0

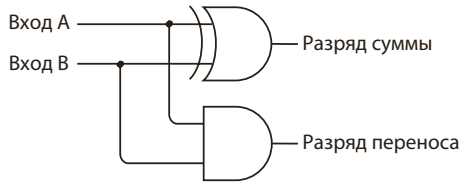
+ перенос	0	1
0	0	0
1	0	1

Для получения этих результатов можно использовать следующие два вентиля.

Искл-ИЛИ	0	1
0	0	1
1	1	0

И	0	1
0	0	0
1	0	1

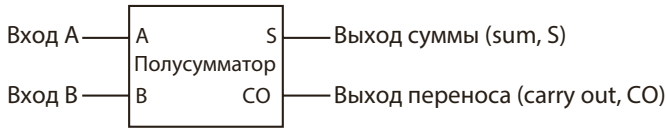
Разряд суммы двух двоичных чисел задается выходом вентиля Искл-ИЛИ, а разряд переноса — выходом вентиля И, поэтому можно комбинировать вентили И и Искл-ИЛИ для сложения двух двоичных цифр A и B.



Вместо многократного перерисовывания вентилей И и Искл-ИЛИ можно просто нарисовать схему, подобную следующей.

---

* В общем случае кружок со стороны выхода означает инверсию результата операции. *Прим. науч. ред.*

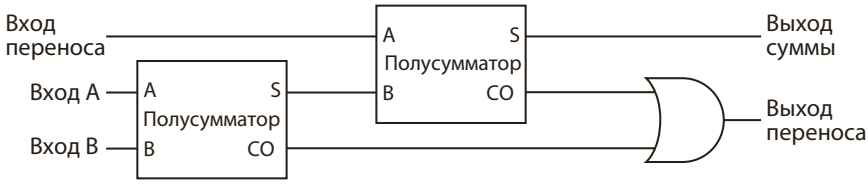


Существует причина, по которой эта схема называется *полусумматором*. Разумеется, она складывает две двоичные цифры и выдает бит суммы и бит переноса. Однако длина подавляющего большинства двоичных чисел превышает один бит. То, что полусумматор не может сделать, так это прибавить возможный бит переноса, получившийся в результате предыдущей операции сложения. Представьте, что складываем два двоичных числа.

```
  1111
+ 1111
------
 11110
```

Мы можем использовать полусумматор только для сложения цифр в правом крайнем столбце: 1 плюс 1 равно 0, 1 переносится. В случае со вторым столбцом справа нам, по сути, нужно сложить *три* двоичные цифры из-за переноса. И это касается всех остальных столбцов. Каждая последующая операция сложения двух двоичных цифр может включать бит переноса из предыдущего столбца.

Для сложения трех двоичных цифр понадобятся два полусумматора и вентиль ИЛИ, соединенные следующим образом.



Чтобы разобраться в этой схеме, начнем со входов A и B первого полусумматора слева. Результат — бит суммы и бит переноса. Эта сумма должна быть добавлена к переносу из предыдущего столбца, поэтому они являются входами для второго полусумматора. Сумма, полученная от второго полусумматора, — окончательная. Два переноса из полусумматоров — входы для вентиля ИЛИ. Может показаться, что здесь нужен второй полусумматор, и такая схема, безусловно, сработала бы. Однако если вы проанализируете все возможности, то обнаружите, что *оба* переноса из двух полусумматоров никогда не равны 1. Вентиля ИЛИ достаточно для их сложения, поскольку он действует так же, как вентиль Искл-ИЛИ, если оба входных сигнала одновременно не равны 1.

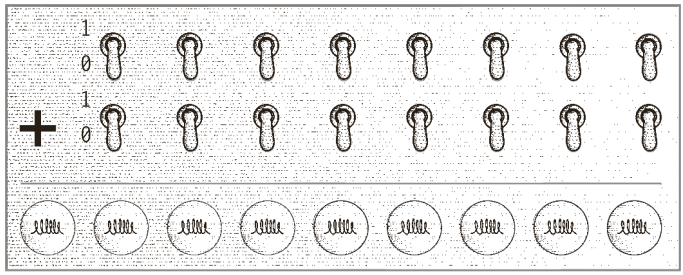
Вместо многократного перерисовывания этой схемы можем просто назвать ее *полным сумматором.*

В следующей таблице представлены все возможные комбинации входов для полного сумматора и результирующие выходы.

Вход A	Вход B	Вход для переноса	Выход для суммы	Выход для переноса
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

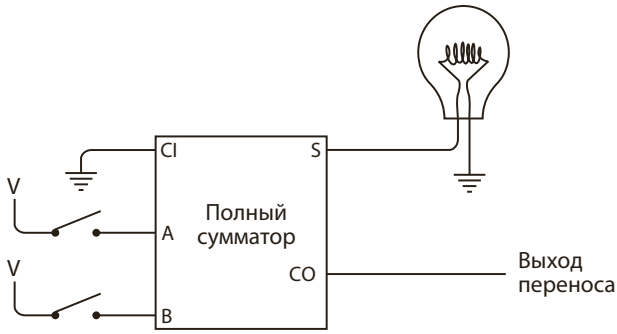
В начале этой главы я сказал, что для создания сумматора потребуются 144 реле. Вот как я это понял: для каждого вентиля И, ИЛИ и И-НЕ требуются по два реле. Таким образом, вентиль Искл-ИЛИ состоит из шести реле. Полусумматор — это вентиль Искл-ИЛИ и вентиль И, поэтому для его создания необходимы восемь реле. Каждый полный сумматор — два полусумматора и вентиль ИЛИ, то есть 18 реле. Нам нужны восемь полных сумматоров для создания 8-битной машины, или 144 реле.

Вспомните наш исходный пульт управления с переключателями и лампочками.



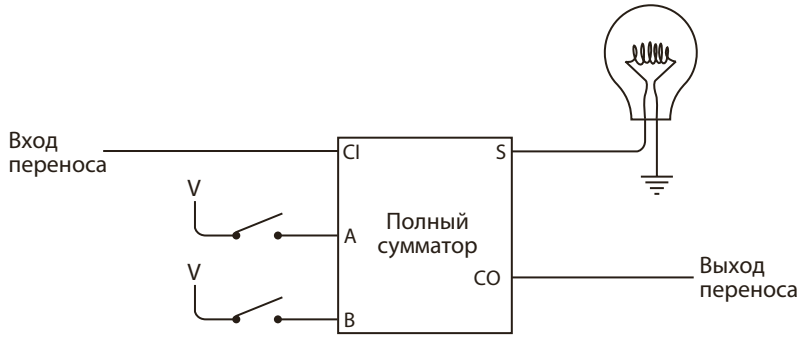
Теперь мы можем начать присоединять переключатели и лампочки к полному сумматору.

Сначала подключим два крайних правых переключателя и крайнюю правую лампочку к полному сумматору.



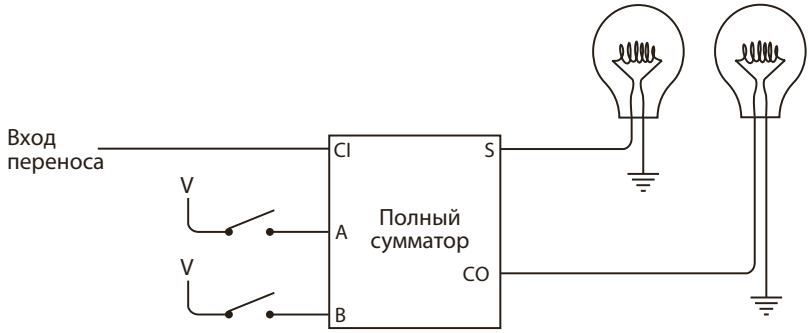
Когда вы начинаете складывать два двоичных числа, первый столбец цифр отличается от остальных тем, что не может содержать бит переноса из предыдущего столбца. В первом столбце нет бита переноса, поэтому вход для переноса полного сумматора соединяется с землей, то есть его значением является 0 бит. Разумеется, *в результате* сложения первой пары двоичных цифр может получиться бит переноса. Этот выход переноса — вход для следующего столбца.

Для следующих двух цифр и лампочки вы используете полный сумматор, подключенный так.



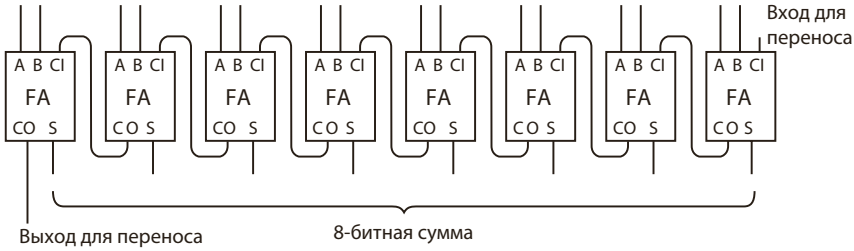
Выход переноса, полученный от первого полного сумматора, является входом для второго полного сумматора. Каждый последующий столбец цифр складывается по той же схеме. Каждый разряд переноса из одного столбца подается на вход для переноса следующего столбца.

Наконец, восьмая и последняя пара переключателей подключена к последнему полному сумматору.

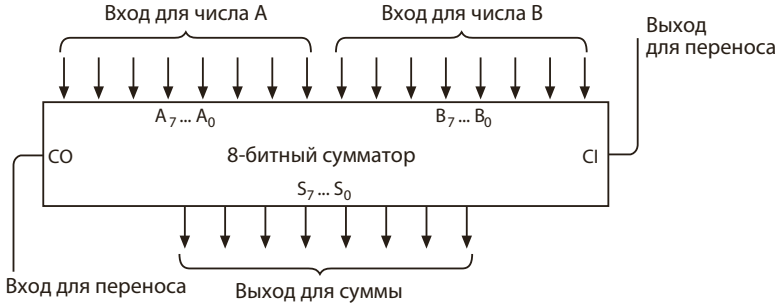


Здесь последний выход для переноса подключен к девятой лампочке.

Вот еще один способ изобразить схему из восьми полных сумматоров (full adder, FA), в которой каждый выход для переноса (CO) подключен к следующему входу для переноса (CI).



Представим единое обозначение 8-битного сумматора, входы обозначим буквами от A_0 до A_7 и от B_0 до B_7, выходы — буквами от S_0 до S_7 (от sum — «сумма»).



Это распространенный способ обозначения отдельных битов многобитного числа. Биты A_0, B_0 и S_0 являются *младшими*, а биты A_7, B_7 и S_7 — *старшими*. Например, вот как с помощью этих букв с индексами можно было бы представить двоичное число 0110 1001.

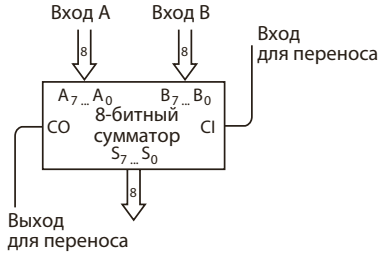
$$\begin{array}{cccccccc} A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Индексы начинаются с 0 и увеличиваются по мере перехода ко все более значимым цифрам, поскольку они соответствуют показателю степени двойки.

$$\begin{array}{cccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

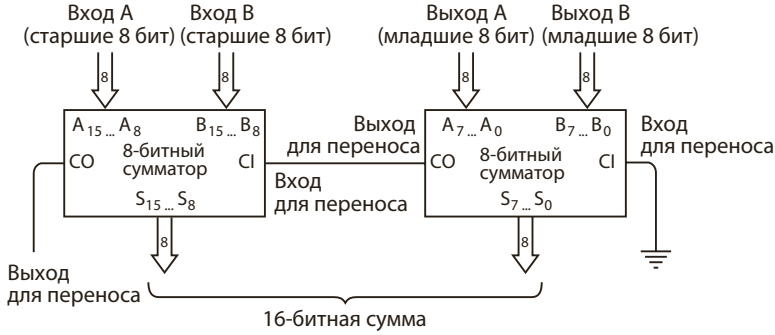
Если вы умножите каждую степень двойки на цифру, расположенную под ней, и сложите результаты, получите десятичный эквивалент числа 0110 1001, который равен 64 + 32 + 8 + 1, или 105.

По-другому 8-битный сумматор можно изобразить так.



Восьмерки внутри стрелок указывают на то, что каждая из них — это группа из восьми отдельных сигналов. Индексы символов $A_7 \ldots A_0$, $B_7 \ldots B_0$ и $S_7 \ldots S_0$ также обозначают восьмиразрядность числа.

Как только соберете один 8-битный сумматор, вы сможете создать второй. Их легко *расположить каскадом*, чтобы сложить два 16-битных числа.



Выход для переноса правого сумматора связан со входом для переноса левого. Левый сумматор в качестве входных значений принимает самые старшие

восемь цифр двух слагаемых и в качестве выходного значения выдает самые старшие восемь цифр.

Теперь вы можете спросить: «Неужели компьютеры *действительно* складывают числа именно так?»

В принципе да. Но не совсем.

Во-первых, сумматоры могут быть быстрее тех, которые мы описали. Если вы посмотрите на то, как работает эта схема, то поймете, что выход переноса от младшей пары цифр необходим для сложения со следующей парой, выход переноса от второй пары цифр — для сложения с третьей парой и т. д. Общая скорость сумматора равна количеству битов, умноженному на скорость одного полного сумматора. Это называется *сквозным переносом*. Быстрые сумматоры используют дополнительные схемы *ускоренного переноса*.

Во-вторых (и это самое главное), компьютерам больше не нужно реле! Однако первые цифровые компьютеры, созданные в начале 1930-х годов, использовали реле, позднее — вакуумные лампы. Современные компьютеры создаются на основе транзисторов. Транзисторы в основном функционируют так же, как и реле, однако (как мы увидим далее) они намного быстрее, компактнее, тише, дешевле и потребляют гораздо меньше энергии. Для построения 8-битного сумматора по-прежнему требуются 144 транзистора (или больше, если вы хотите заменить сквозной перенос схемой ускоренного переноса), однако при этом размер схемы микроскопический.

Глава 13
А как насчет вычитания?

Убедившись в том, что реле действительно можно соединить для сложения двоичных чисел, зададимся вопросом: «А как насчет вычитания?» Не бойтесь показаться смешным, задавая его. На самом деле вы довольно проницательны. Сложение и вычитание в некотором смысле дополняют друг друга, однако механика у этих операций разная. Сложение предполагает последовательное продвижение от крайнего правого столбца цифр до крайнего левого. Каждое значение, перенесенное из одного столбца, прибавляется к следующему. При вычитании мы ничего *не переносим*; мы *заимствуем*, а это действие предполагает использование довольно запутанного механизма.

Давайте рассмотрим типичную задачу на вычитание с заимствованием.

```
 253
−176
 ???
```

Начнем решение с крайнего правого столбца. Сначала мы замечаем, что 6 больше 3, поэтому нам нужно занять 1 у 5, а затем вычесть 6 из 13, в результате чего получается 7. Мы помним, что заняли 1 у 5, поэтому вместо 5 имеем 4, что меньше 7, поэтому занимаем 1 у 2, вычитаем 7 из 14 и получаем 7. Мы заняли 1 у 2, поэтому у нас есть только единица, из которой вычитаем 1 и получаем 0. Наш ответ — 77.

```
 253
−176
  77
```

Как же заставить группу логических вентилей следовать такой странной логике?

Не будем даже пытаться. Вместо этого используем небольшой трюк, позволяющий вычитать *без* заимствования. Это порадовало бы Полония («Не занимай и не ссужай») и всех остальных. Кроме того, подробное рассмотрение процесса вычитания полезно, поскольку напрямую связано с использованием двоичных кодов для хранения в компьютерах отрицательных чисел.

Числа, участвующие в операции вычитания, называются *уменьшаемым* и *вычитаемым*. Вычитаемое вычитается из уменьшаемого, результатом является *разность*.

```
  Уменьшаемое
− Вычитаемое
  ----------
  Разность
```

Чтобы произвести вычитание без заимствования, сначала нужно вычесть вычитаемое из 999, а не из уменьшаемого.

```
  999
− 176
  ---
  823
```

В данном случае используем число 999, поскольку числа, участвующие в операции, состоят из трех цифр. Если бы они были четырехзначными, мы бы использовали число 9999. При вычитании числа из строки девяток получаем число, называемое *дополнением до девяти*. Дополнение числа 176 до девяти — 823. Это работает и в обратную сторону: дополнение числа 823 до девяти — 176. Вся прелесть в том, что вне зависимости от значения вычитаемого вычисление его дополнения до девяти *никогда не требует заимствования*.

После вычисления дополнения вычитаемого до девяти нужно прибавить к нему исходное уменьшаемое.

```
  253
+ 823
  ----
  1076
```

Наконец, прибавить 1 и вычесть 1000.

```
  1076
+    1
− 1000
  ----
    77
```

Вот и всё. Мы получили такой же результат, как и раньше, ни разу не прибегнув к заимствованию.

Почему это работает? Исходная задача на вычитание такова:

$$253 - 176.$$

Если к этому выражению прибавить и вычесть любое число, результат останется прежним. Так что давайте прибавим и вычтем 1000:

$$253 - 176 + 1000 - 1000.$$

Выражение эквивалентно следующему:

$$253 - 176 + 999 + 1 - 1000.$$

Теперь числа можно перегруппировать:

$$253 + (999 - 176) + 1 - 1000.$$

Это соответствует вычислению, которое я продемонстрировал с использованием дополнения до девяти. Мы заменили одно вычитание двумя вычитаниями и двумя сложениями, избавившись при этом от всех нежелательных заимствований.

А если вычитаемое больше уменьшаемого? Рассмотрим такой пример.

$$\begin{array}{r} 176 \\ -\ 253 \\ \hline ??? \end{array}$$

Обычно вы смотрите на подобную задачу и думаете: «Хм, вычитаемое больше уменьшаемого, поэтому придется поменять числа местами, выполнить вычитание и не забыть о том, что результат будет отрицательным». Вы можете произвести перестановку чисел в голове и записать ответ следующим образом.

$$\begin{array}{r} 176 \\ -\ 253 \\ \hline -77 \end{array}$$

Процесс выполнения данного расчета без заимствований несколько отличается от предыдущего примера. Как и раньше, мы начинаем с вычитания вычитаемого (253) из 999 для получения дополнения до девяти.

$$\begin{array}{r} 999 \\ -\ 253 \\ \hline 746 \end{array}$$

Теперь прибавим дополнение до девяти к исходному уменьшаемому.

$$\begin{array}{r} 176 \\ +\ 746 \\ \hline 922 \end{array}$$

На этом этапе в более раннем примере для получения окончательного результата мы могли прибавить 1 и вычесть 1000. Однако в данном случае эта стратегия не сработала бы, поскольку нам пришлось бы вычесть 1000 из 923, что в действительности потребовало бы вычесть 923 из 1000, и без заимствований мы бы не обошлись.

Вместо этого, по аналогии с прибавлением 999, вычтем 999.

$$\begin{array}{r} 922 \\ -\ 999 \\ \hline ??? \end{array}$$

Сразу становится очевидным, что наш ответ будет отрицательным, поэтому следует поменять числа местами и вычесть 922 из 999. Это опять же не требует заимствований, а ответ совпадает с ожидаемым.

$$\begin{array}{r} 922 \\ -\ 999 \\ \hline -77 \end{array}$$

Этот метод применим и к двоичным числам, работать с которыми оказывается проще, чем с десятичными. Давайте посмотрим, как это работает.

Вот исходная задача на вычитание.

```
  253
- 176
-----
  ???
```

После преобразования чисел в двоичные получаем следующую задачу.

```
  11111101
- 10110000
----------
  ????????
```

Шаг 1. Вычесть вычитаемое из 11111111 (что соответствует 255).

```
  11111111
- 10110000
----------
  01001111
```

При работе с десятичными числами вычитаемое вычиталось из строки девяток, а результат назывался дополнением до девяти. При работе с двоичными числами вычитаемое вычитается из строки единиц, результат — *дополнение до единицы*. Заметьте, что нам на самом деле не нужно выполнять вычитание, чтобы вычислить дополнение до единицы, поскольку каждый 0 в исходном числе превращается в 1 в дополнении до единицы, а каждая 1 превращается в 0. По этой причине дополнение до единицы иногда также называется *отрицанием* или *инверсией*. (Сейчас вы, вероятно, вспомнили о том, что в главе 11 мы конструировали устройство, называемое инвертором, которое меняло 0 на 1, а 1 на 0.)

Шаг 2. Прибавить дополнение вычитаемого до единицы к уменьшаемому.

```
  11111101
+ 01001111
----------
  101001100
```

Шаг 3. Прибавить к результату 1.

```
  101001100
+           1
-------------
  101001101
```

Шаг 4. Вычесть 100000000 (что соответствует 256).

```
  101001101
− 100000000
-----------
    1001101
```

Результат эквивалентен десятичному числу 77.

Давайте попробуем еще раз, поменяв числа местами. В десятичной системе счисления задача на вычитание выглядит так.

```
  176
− 253
-----
  ???
```

В двоичной системе — так.

```
  10110000
− 11111101
----------
  ????????
```

Шаг 1. Вычесть вычитаемое из 11111111, чтобы получить дополнение до единицы.

```
  11111111
− 11111101
----------
  00000010
```

Шаг 2. Прибавить дополнение вычитаемого до единицы к уменьшаемому.

```
  10110000
+ 00000010
----------
  10110010
```

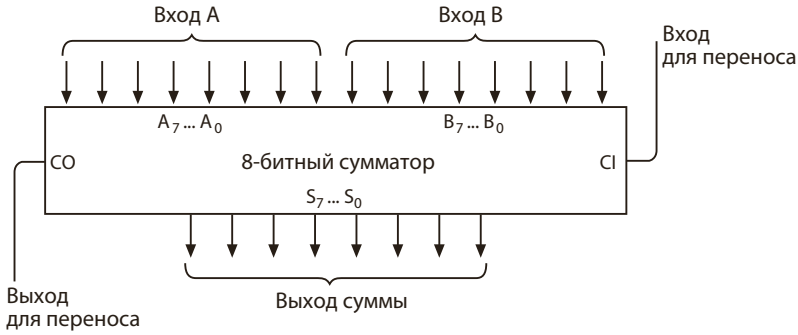
Теперь нужно как-то вычесть из результата число 11111111. Когда исходное вычитаемое меньше уменьшаемого, для этого достаточно прибавить 1 и вычесть 100000000. Однако эту операцию нельзя выполнить без заимствования. Вместо этого мы вычитаем результат из числа 11111111.

```
  11111111
− 10110010
----------
  01001101
```

Опять же, такая стратегия на самом деле означает, что для получения результата мы выполняем простое инвертирование. Ответ снова равен 77, а на самом деле –77.

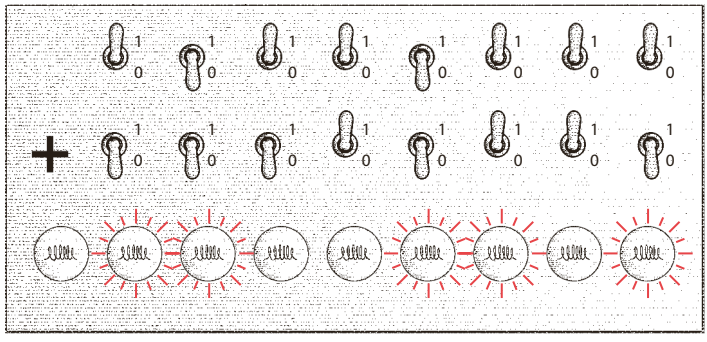
Теперь у нас есть необходимые знания для оснащения собранного в предыдущей главе сумматора функцией вычитания. Чтобы *не слишком* усложнять эту счетную машину, сделаем так, чтобы она выполняла вычитание только тогда, когда вычитаемое меньше уменьшаемого, когда в результате получается положительное число.

Основой этой счетной машины являлся 8-разрядный сумматор, собранный из логических вентилей.



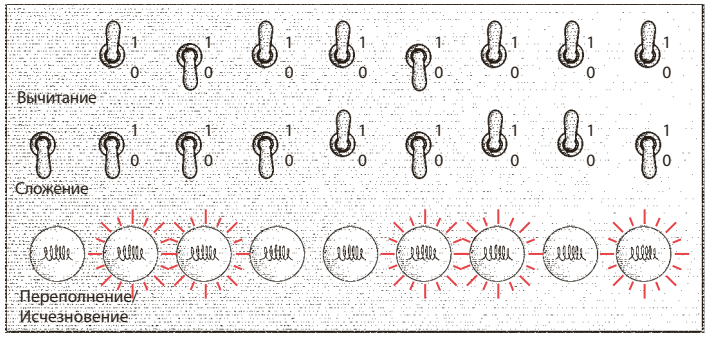
Вероятно, вы помните, что входы с A_0 по A_7 и с B_0 по B_7 были подключены к переключателям, с помощью которых вводились два 8-битных числа, подлежащие сложению. Вход для переноса соединялся с землей, выходы с S_0 по S_7 — с восемью лампочками, отображающими результат сложения. Поскольку в итоге могло получиться 9-битное значение, выход для переноса был подключен к девятой лампочке.

Пульт управления выглядел так.



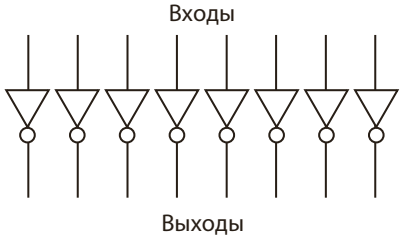
Положения переключателей на этом изображении соответствуют сложению чисел 183 (10110111) и 22 (00010110), в результате которого получается 205, или 11001101, что и отражено рядом лампочек.

Новый пульт управления для сложения и вычитания двух 8-битных чисел имеет несколько иной вид. Он предусматривает дополнительный переключатель, позволяющий выбрать между сложением и вычитанием.



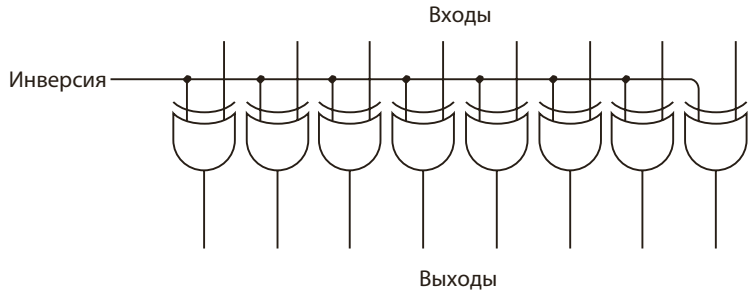
Подписи подсказывают, что размыкание этого переключателя соответствует сложению, а замыкание — вычитанию. Кроме того, для отображения результатов используются только восемь крайних лампочек справа. Девятая лампочка обозначена словами «Переполнение/Исчезновение». Она загорается, когда полученное число не может быть представлено восемью лампочками. Так происходит, если в результате сложения получается число, превышающее 255 (это называется переполнением), или если результат вычитания — отрицательное число — исчезновение порядка, то есть если вычитаемое больше уменьшаемого.

Главное нововведение в счетной машине — схема, которая вычисляет дополнение до единицы для 8-битного числа. Напомним, что вычисление дополнения до единицы эквивалентно инвертированию битов, поэтому устройство для произведения этой операции могло бы состоять просто из восьми инверторов.



Проблема этой схемы заключается в том, что она *всегда* инвертирует поступающие в нее биты. Наша цель — создать машину, которая выполняет как

сложение, так и вычитание, поэтому эта схема должна инвертировать биты только при выполнении вычитания. Вот более подходящая схема.



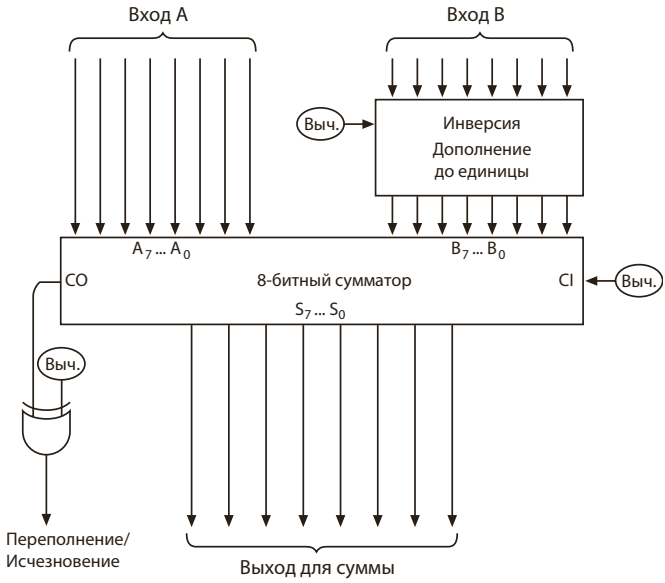
Сигнал «Инверсия» поступает на каждый из восьми вентилей Искл-ИЛИ (исключающее ИЛИ). Напомним, вентиль Искл-ИЛИ работает так.

Искл-ИЛИ	**0**	**1**
0	0	1
1	1	0

Если сигнал «Инверсия» равен 0, то сигналы на восьми выходах вентилей Искл-ИЛИ совпадают с сигналами на их входах. Например, если на вход подается значение 01100001, то выходным значением также является 01100001. Если сигнал «Инверсия» равен 1, то восемь входных сигналов будут инвертированы. Если на вход подается значение 01100001, то выходное — 10011110.

Давайте поместим эти восемь вентилей Искл-ИЛИ в прямоугольник под названием «Дополнение до единицы».

Теперь устройство для дополнения числа до единицы, 8-битный сумматор и последний вентиль Искл-ИЛИ можно объединить в схему.



Обратите внимание на три сигнала, обозначенные как «Выч.». Они поступают от переключателя «Сложение/вычитание». Этот сигнал равен 0, если необходимо выполнить сложение, и 1, если вычитание. При вычитании сигналы на входах B (второй ряд переключателей) инвертируются схемой «Дополнение до единицы» перед попаданием в сумматор. Кроме того, при вычитании к результату сложения прибавляется 1 за счет подачи на вход сумматора CI (вход для переноса) значения 1. При сложении схема «Дополнение до единицы» не оказывает никакого эффекта, а вход CI равен 0.

Сигнал «Выч.» и выходной сигнал сумматора CO (выход для переноса) также подаются на вход вентиля Искл-ИЛИ, который используется для включения лампочки «Переполнение/Исчезновение». Если сигнал «Выч.» равен 0 (выполняется сложение), лампочка будет гореть при выходном сигнале сумматора CO, равном 1. Это означает, что результат сложения превышает 255.

При выполнении вычитания, когда вычитаемое (переключатели B) меньше уменьшаемого (переключатели A), выходной сигнал сумматора CO бывает равен 1. Это нормально и означает, что на последнем этапе необходимо вычесть 100000000. Таким образом, лампа «Переполнение/Исчезновение» горит только тогда, когда выход сумматора CO равен 0, то есть вычитаемое больше уменьшаемого и результат отрицательный. Описанная выше машина не предназначена для отображения отрицательных чисел.

Сейчас вы, вероятно, рады, что спросили: «А как насчет вычитания?»

В этой главе я говорил об отрицательных числах, но все еще не показал, как выглядят отрицательные двоичные числа. Вы можете предположить, что

традиционный знак «–» используется с двоичными числами так же, как и с десятичными. Например, число –77 в двоичном формате записывается –1001101. Конечно, вы можете так его записать, однако одна из целей использования двоичных чисел заключается в том, чтобы представлять *всё* с помощью нулей и единиц, включая такие символы, как «–» перед отрицательным числом.

Можно просто использовать еще один бит для знака «–». Его значение 1 может соответствовать отрицательному числу, а 0 — положительному, и это будет работать, хотя мало что даст. Существует еще одно решение для представления отрицательных чисел, которое также предусматривает простой способ сложения отрицательных и положительных чисел. Недостаток этого метода — необходимость заранее решить, сколько цифр требуется для представления чисел, с которыми мы будем работать.

Давайте подумаем. Преимущество обычного способа представления положительных и отрицательных чисел заключается в том, что они могут иметь бесконечную длину. Мы представляем 0 как точку, от которой в одну сторону в бесконечность уходят положительные числа, а в другую — отрицательные.

… –1 000 000 –999 999 … –3 –2 –1 0 1 2 3 … 999 999 1 000 000 …

Однако представьте, что нам не нужно бесконечное количество чисел: с самого начала известно, что каждое число, которое встретится, будет находиться в определенном диапазоне.

Рассмотрим чековый банковский счет, в котором мы иногда сталкиваемся с отрицательными числами. Предположим, что баланс нашего счета никогда не превышал 500 долларов, и банк установил лимит перерасхода на те же 500 долларов. Это значит, что баланс нашего счета всегда находится в диапазоне от 499 до –500 долларов. При этом мы никогда не кладем на счет более 499 долларов, никогда не выписываем чек на сумму более 500 долларов и имеем дело только с долларами, центы же не учитываем.

Этот набор условий указывает, что при использовании нашего счета мы имеем дело с числами в диапазоне от –500 до 499. Это всего 1000 чисел. Такое ограничение подразумевает, что для представления *всех* нужных чисел мы должны использовать только три десятичные цифры и обходиться без знака «–». Хитрость в том, что нам не нужны положительные числа от 500 до 999, поскольку мы уже решили, что максимальным положительным числом для нас является 499. Таким образом, трехзначные числа от 500 до 999 фактически можно использовать для представления отрицательных чисел. Вот как это работает.

Вместо –500 используем 500.
Вместо –499 используем 501.
Вместо –498 используем 502.
…
Вместо –2 используем 998.
Вместо –1 используем 999.
Вместо 0 используем 000.
Вместо 1 используем 001.
Вместо 2 используем 002.
…
Вместо 497 используем 497.
Вместо 498 используем 498.
Вместо 499 используем 499.

Другими словами, каждое трехзначное число, которое начинается с цифры 5, 6, 7, 8 или 9, фактически является отрицательным. Вместо того чтобы записывать эти числа как:

–500 –499 –498 … –4 –3 –2 –1 0 1 2 3 4 … 497 498 499,

запишем их:

500 501 502 … 996 997 998 999 000 001 002 003 004 … 497 498 499.

Обратите внимание: числа идут по кругу. Наименьшее отрицательное число (500) как бы следует за наибольшим положительным числом (499). А число 999 (которое фактически равно –1) на единицу меньше нуля. Если мы прибавим 1 к 999, то в обычных условиях получим 1000. Но поскольку мы имеем дело только с тремя цифрами, в результате будет 000.

Такой способ записи отрицательных чисел называется *дополнением до десяти*. Чтобы преобразовать трехзначное отрицательное число в дополнение до десяти, вычитаем его из 999 и прибавляем 1. Другими словами, дополнение до десяти — это дополнение до девяти плюс один. Например, чтобы найти дополнение числа –255 до десяти, нужно вычесть его из 999, получив 744, а затем прибавить 1, что в результате даст 745.

Вероятно, вы слышали утверждение, что вычитание — это просто прибавление отрицательных чисел. На что вы, вероятно, отвечали: «Да, но числа все равно приходится *вычитать*». Используя дополнение до десяти, вы вообще ничего не вычитаете. Все сводится к сложению.

Предположим, у вас на счету 143 доллара. Вы выписываете чек на 78 долларов. Это означает, что вы должны прибавить –78 к 143. Дополнение числа –78 до десяти равно 999 – 078 + 1, то есть 922. Таким образом, ваш баланс теперь составляет 143 + 922, или 65 долларов (без учета переполнения). Если после этого вы выпишете чек на 150 долларов, придется прибавить число –150,

дополнение которого до десяти равно 850. Итак, прибавим 850 к предыдущему балансу 065 и получим 915 — новый баланс. Это число фактически эквивалентно –85.

В двоичном формате подобная система называется *дополнением до двух*. Предположим, что мы работаем с 8-битными числами в диапазоне от 00000000 до 11111111, которые соответствуют десятичным числам от 0 до 255. Если вам требуется использовать отрицательные числа, то каждое 8-битное число, начинающееся с 1, фактически будет отрицательным:

Двоичное число	**Десятичное число**
10000000	–128
10000001	–127
10000010	–126
10000011	–125
…	…
11111101	–3
11111110	–2
11111111	–1
00000000	0
00000001	1
00000010	2
…	…
01111100	124
01111101	125
01111110	126
01111111	127

Теперь вы можете представить числа в диапазоне от –128 до +127. Старший значащий бит (крайний слева) называется *знаковым разрядом*. Знаковый разряд равен 1 для отрицательных чисел и 0 — для положительных.

Чтобы вычислить дополнение числа до двух, сначала вычислите его дополнение до единицы, а затем прибавьте 1. Это эквивалентно инвертированию всех цифр и прибавлению 1. Например, десятичное число 125 в двоичном формате выражается как 01111101. Чтобы выразить число –125 в виде дополнения до двух, сначала инвертируем цифры 01111101 для получения 10000010, а затем прибавим 1, в результате чего получим 10000011. Вы можете проверить результат по приведенной выше таблице. Для выполнения обратной операции повторите те же действия — инвертируйте все биты и прибавьте 1.

Эта система позволяет выражать положительные и отрицательные числа, не используя знак «–». Кроме того, с ее помощью можно складывать положительные и отрицательные числа, руководствуясь только правилами сложения. Например, сложим двоичные эквиваленты чисел –127 и 124. Это просто, если использовать в качестве шпаргалки предыдущую таблицу.

```
  10000001
– 01111100
----------
  11111101
```

Результат эквивалентен числу –3 в десятичной системе счисления.

В данном случае нужно следить за условиями переполнения и исчезновения разряда. Такое может произойти, когда в результате сложения получается число больше 127 или меньше –128. Например, вы прибавляете 125 к 125.

```
  01111101
– 01111101
----------
  11111010
```

Поскольку старший бит равен 1, результат следует интерпретировать в качестве отрицательного числа, которое соответствует –6 в десятичной системе счисления. Что-то подобное происходит и при сложении чисел –125 и –125.

```
  10000011
– 10000011
----------
 100000110
```

С самого начала мы решили ограничиться 8-битными числами, поэтому необходимо проигнорировать крайнюю левую цифру. Правые восемь бит эквивалентны числу +6.

Вообще, результат сложения положительных и отрицательных чисел не является верным, если знаковые разряды двух операндов одинаковы, а знаковый разряд результата отличается.

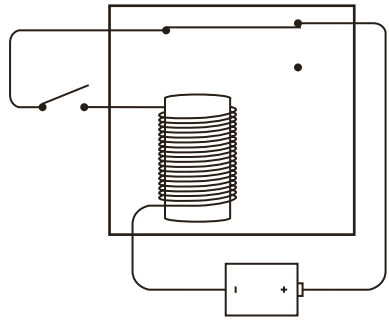
Теперь у нас есть два разных способа использования двоичных чисел. Двоичное число может быть *со знаком (знаковым)* или *без знака (беззнаковым)*. Восьмибитные числа без знака находятся в диапазоне от 0 до 255, 8-битные числа со знаком — в диапазоне от –128 до 127. По самим числам невозможно понять, сопровождаются они знаком или нет. Например, кто-то говорит: «У меня есть 8-битное двоичное число, значение которого равно 10110110. Каков его десятичный эквивалент?» Сначала вы должны спросить: «Это число со знаком или без знака? В зависимости от этого ответом может быть либо число –74, либо 182».

В этом и состоит проблема с битами: они всего лишь нули и единицы, которые ничего не говорят *о себе*.

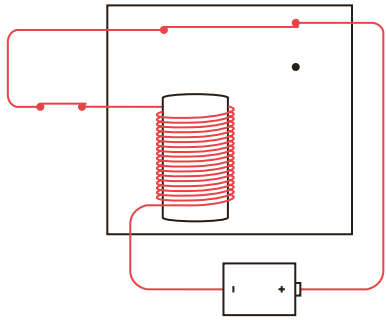
Глава 14
Обратная связь и триггеры

Известно, что электричество приводит предметы в движение. Оглядевшись по сторонам в обычном доме, можно увидеть электрические двигатели в различных устройствах, например в часах, вентиляторах, кухонных комбайнах, проигрывателях компакт-дисков. Электричество также заставляет вибрировать конусы в динамиках, благодаря чему стереосистемы и телевизоры воспроизводят звуки, речь и музыку. Однако самый простой и изящный пример того, как электричество приводит предметы в движение, вероятно, иллюстрируется классом устройств, которые быстро исчезают по мере их замены электронными аналогами. Я имею в виду уже ставшие раритетом электрические зуммеры и звонки.

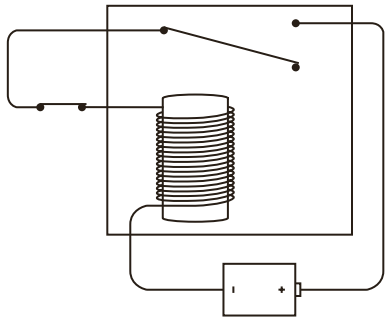
Рассмотрим реле, соединенное с переключателем и батарейкой следующим образом.



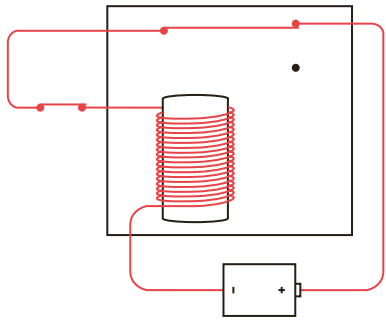
Неудивительно, если эта схема кажется немного странной. Мы еще не встречались с таким способом подключения. Обычно реле подключается так, что вход отделен от выхода. Здесь все закольцовано. Замыкание переключателя делает цепь непрерывной.



Ток в замкнутой цепи заставляет электромагнит притягивать гибкую полоску.

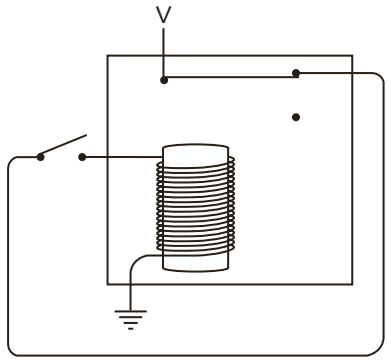


Когда полоска меняет свое положение, цепь размыкается, поэтому электромагнит теряет свои магнитные свойства, а полоска возвращается на свое место.

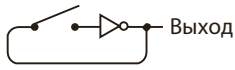


А это, конечно же, снова замыкает схему. Происходит следующее: до тех пор, пока переключатель замкнут, металлическая полоска двигается назад и вперед, поочередно замыкая и размыкая цепь, что, скорее всего, сопровождается звуком. Устройство, издающее дребезжащий звук, называется зуммером. Если вы присоедините к нему молоточек и разместите рядом чашечку, появится электрический звонок.

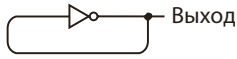
Чтобы сделать зуммер, можно выбрать один из двух способов подключения этого реле. Вот еще одна схема подключения с общепринятыми символами для обозначения источника питания и земли.



Возможно, в этой схеме вы узнали инвертор, описанный в главе 11, поэтому ее можно изобразить проще.



Как вы помните, выход инвертора — 1, если вход — 0, и выход — 0, если вход — 1. Замыкание переключателя в этой цепи приводит к поочередному размыканию и замыканию реле в инверторе. Постоянную работу инвертора может обеспечить и схема без переключателя.



Может показаться, что эта иллюстрация противоречит логике, поскольку выход инвертора должен располагаться напротив входа, однако в данном случае выход *является* входом! Имейте в виду, что инвертор на самом деле просто реле, которому требуется немного времени для перехода из одного состояния в другое. Так что даже если вход равен выходу, вскоре выход станет противоположен входу (что, конечно, приведет к изменению входного сигнала и т. д.).

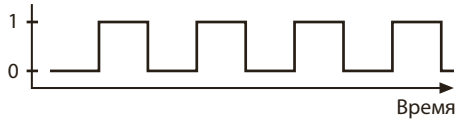
Чему равен выход этой цепи? Его значение быстро меняется между наличием и отсутствием напряжения. Можно сказать, *значение выхода быстро чередуется между 0 и 1*.

Такая цепь называется *осциллятором*. По сути она отличается от всех устройств, которые мы рассматривали ранее. Все виденные нами схемы изменяли свое состояние только при вмешательстве человека, который замыкал и размыкал переключатель. Однако осциллятор не нуждается в человеке, он работает сам по себе.

Разумеется, в отрыве от окружения осциллятор не очень полезен. Далее в этой и последующих главах мы увидим, что такая схема, подключенная к другим схемам, является важной частью автоматизации. Во всех компьютерах присутствует некий осциллятор, обеспечивающий синхронную работу остальных частей.

Значение выхода осциллятора чередуется между 0 и 1. Часто этот факт изображается с помощью диаграммы.

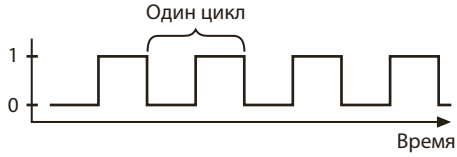
Эту диаграмму можно воспринимать как график, по горизонтальной оси которого отложено время, а по вертикальной — выходные значения 0 и 1.



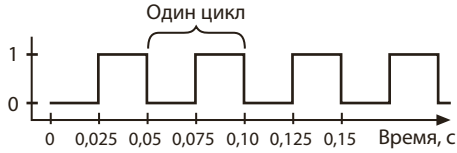
Все это говорит о том, что с течением времени выходное значение осциллятора регулярно изменяется с 0 на 1. По этой причине осциллятор иногда называют *часами*, поскольку он позволяет определить время: достаточно подсчитывать количество колебаний.

Как быстро будет работать осциллятор? Насколько быстро будет вибрировать металлический контакт реле? Сколько раз в секунду? Очевидно, это зависит от конструкции реле. Легко представить большое, неуклюжее реле, которое медленно замыкает и размыкает контакт, и небольшое легкое реле с быстро вибрирующим контактом.

Цикл осциллятора — интервал, в течение которого его выход изменяется, после чего возвращается к исходному значению.



Время, которое занимает один цикл, называется *периодом* осциллятора. Предположим, период нашего осциллятора равен 0,05 секунды. Вдоль горизонтальной оси мы можем отложить время в секундах начиная с некоторого произвольно выбранного нулевого момента.



Частота осциллятора равна единице, поделенной на период. В данном примере, если период осциллятора составляет 0,05 секунды, его частота 1 / 0,05 = 20 *колебаний в секунду*. Выход осциллятора изменяется и возвращается к исходному значению 20 раз в секунду.

Количество колебаний в секунду — такой же логичный термин, как *количество километров в час*, *килограммов на квадратный метр* или *калорий на порцию*, однако он больше не используется. В память о Генрихе Рудольфе Герце (1857–1894), который первым передал и принял радиоволны, говорят «*герц*». Сначала это слово начали применять в Германии в 1920-х годах, а затем за несколько десятилетий термин прижился и в других странах.

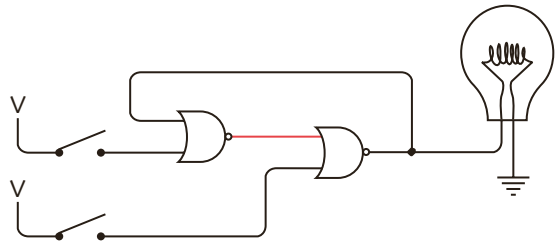
Таким образом, можно сказать, что наш осциллятор имеет частоту 20 герц, или 20 Гц (сокращенно).

Конечно, мы произвольно указали частоту одного конкретного осциллятора. В конце этой главы мы сможем соорудить то, что позволит фактически измерить данный параметр.

Чтобы начать работу над этим устройством, давайте рассмотрим пару вентилей ИЛИ-НЕ, соединенных определенным образом. Вероятно, вы помните, что на выходе вентиля ИЛИ-НЕ есть напряжение, только если напряжения нет ни на одном из его входов.

ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

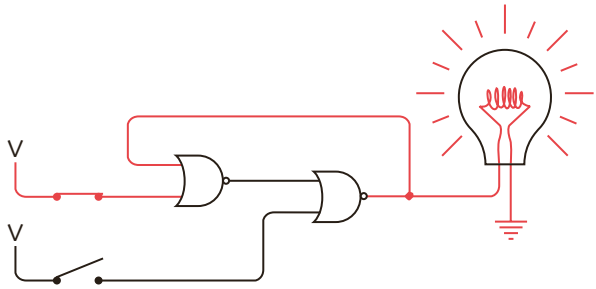
Вот схема с двумя вентилями ИЛИ-НЕ, двумя переключателями и лампочкой.



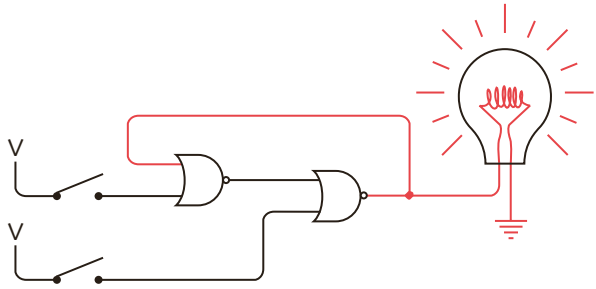
Обратите внимание на необычную схему проводки: выход левого вентиля ИЛИ-НЕ — это вход правого вентиля ИЛИ-НЕ, а выход правого вентиля ИЛИ-НЕ — вход левого вентиля ИЛИ-НЕ. Такое соединение называется *обратной связью*. Действительно, как и в случае с осциллятором, выход становится

входом. Эта особенность будет характерна для большинства схем, приведенных в данной главе.

Сначала ток в этой цепи будет течь только от выхода левого вентиля ИЛИ-НЕ. Это связано с тем, что оба входа данного вентиля равны 0. Теперь замкните верхний переключатель. Выход левого вентиля ИЛИ-НЕ становится равным 0, а это значит, что выход правого вентиля ИЛИ-НЕ равен 1, и лампочка загорается.

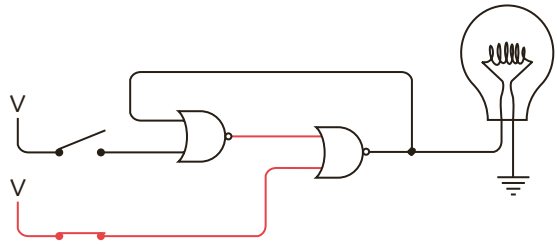


Волшебство наблюдается, когда вы размыкаете верхний переключатель. Поскольку выход вентиля ИЛИ-НЕ — 0, если один из входов — 1, выход левого вентиля ИЛИ-НЕ не изменяется, и лампочка не гаснет.

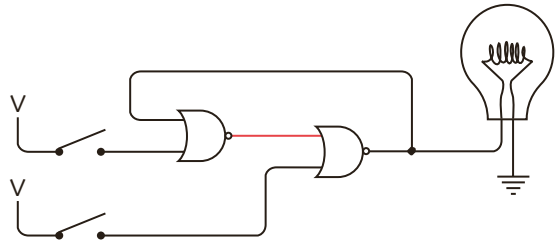


Не кажется ли вам это странным? Оба переключателя разомкнуты, как и на первом рисунке, однако теперь лампочка горит. Эта ситуация, безусловно, отличается от тех, что мы видели ранее. Обычно выход цепи зависит только от ее входов. Похоже, в данном случае это не так. Более того, на этом этапе вы можете замкнуть и разомкнуть верхний переключатель, и лампочка не погаснет. Этот переключатель больше не влияет на цепь, поскольку выход левого вентиля ИЛИ-НЕ остается равным 0.

Теперь замкните нижний переключатель. Поскольку один из входов правого вентиля ИЛИ-НЕ равен 1, выход становится равным 0, лампочка гаснет. При этом выход левого вентиля ИЛИ-НЕ — 1.



Если вы разомкнете нижний переключатель, лампочка останется выключенной.



Мы вернулись туда, откуда начали. Теперь вы можете замкнуть и разомкнуть нижний переключатель, не повлияв на лампочку. Подытожим:

— при замыкании верхнего переключателя лампочка загорается и остается гореть при размыкании верхнего переключателя;
— при замыкании нижнего переключателя лампочка гаснет и не загорается при размыкании нижнего переключателя.

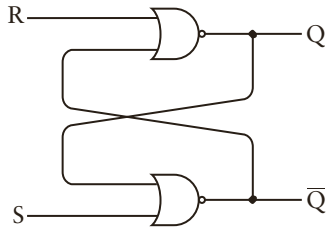
Странность этой схемы заключается в том, что, когда оба переключателя разомкнуты, иногда лампочка горит, иногда — нет. Можно сказать, что эта схема имеет два *устойчивых состояния*, когда оба переключателя разомкнуты. Называется такая схема *триггером*, и его история началась в 1918 году с работы английского радиофизика Уильяма Генри Эклза (1875–1966) и Фрэнка Джордана (о котором мало что известно).

Триггер *сохраняет информацию*, «помнит». В частности, показанный ранее триггер помнит, какой переключатель был замкнут последним. Если вы столкнулись с таким триггером и видите, что лампочка горит, можете предположить, что последним был замкнут верхний переключатель; если лампочка не горит — нижний.

Триггер похож на качели, которые имеют два устойчивых состояния, никогда подолгу не задерживаются в неустойчивом среднем положении. Глядя на качели, вы всегда можете сказать, в какую сторону они качнулись в последний раз.

Возможно, это не является очевидным, однако триггеры весьма полезны. Они обеспечивают память схемы, сохраняющую историю того, что произошло ранее. Представьте, что значит считать, не обладая памятью. В этом случае вы не знаете, какое число задумали, какое число следует к нему прибавить! Точно так же *схема*, которая производит подсчет (описанная далее), требует наличия триггеров.

Существуют два различных типа триггеров. Тот, что я показал выше, является самым простым и называется *RS-триггером* (Reset/Set, сброс/установка). Два вентиля ИЛИ-НЕ чаще всего изображаются и обозначаются так, как показано на диаграмме, для придания им симметричного вида.



Выход, который мы использовали для лампочки, традиционно называется Q (от английского quit — «выход»). Кроме того, существует второй выход $\overline{Q}$, который является электрически противоположным выходу Q, то есть инверсией Q. Если Q равен 0, то Q равен 1, и наоборот. Два входа, S и R, используются для *установки* (set) и *сброса* (reset). Вы можете думать об этих действиях так: «установить значение Q на 1» и «сбросить значение Q на 0». Когда S равно 1 (что соответствует замыканию верхнего переключателя на приведенной раннее диаграмме), выход Q становится равным 1, а выход $\overline{Q}$ — 0. Когда R равен 1 (что соответствует замыканию нижнего переключателя на приведенной ранее диаграмме), выход Q становится равным 0, а выход $\overline{Q}$ — 1. Когда оба входа равны 0, выход указывает на то, являлось ли последним действием установка или сброс значения Q. Результаты работы этой схемы приведены в следующей таблице.

Входы		**Выходы**	
S	**R**	**Q**	**$\overline{Q}$**
1	0	1	0
0	1	0	1
0	0	Q	$\overline{Q}$
1	1	Запрещено	

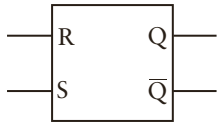
Эта схема называется *функциональной таблицей*, или *таблицей логики,* или *таблицей истинности*. В ней показаны значения выходов, которые являются результатом определенных комбинаций входов. Поскольку у RS-триггера есть только два входа, существует всего четыре комбинации входов. Они соответствуют четырем строкам таблицы.

Обратите внимание на вторую строку снизу, когда значения входов S и R равны 0: выходы обозначены символами Q и $\overline{Q}$, следовательно, значения выходов Q и $\overline{Q}$ остаются такими, какими они были до того, как значения обоих входов S и R стали равны 0. Последняя строка таблицы говорит о том, что ситуация, при которой значения входов S и R равны 1, *запрещена*. Это не означает, что вас могут за это арестовать, однако если в этой схеме оба входа будут 1, то оба выхода — 0, что нарушает условие, согласно которому выход Q противоположен выходу $\overline{Q}$. Так что при создании схемы, в которой используется RS-триггер, избегайте ситуаций, когда входы S и R равны 1.

От себя добавлю: обычно таблицы истинности триггеров изображают с учетом предыдущего состояния Q(t – 1), соответственно, актуальное состояние на выходе триггера — Q(t). Таким образом, для RS-триггера можно составить следующую таблицу.

Входы			**Выходы**	
Q(t – 1)	**S**	**R**	**Q(t)**	**Q(t)**
0	1	0	1	0
1	1	0	1	0
0	0	1	0	1
1	0	1	0	1
0	0	0	0	1
1	0	0	1	0
0	1	1	Запрещено	
1	1	1	Запрещено	

RS-триггер часто изображается в виде небольшого прямоугольника с двумя входами и двумя выходами, обозначенными, как показано ниже.



RS-триггер, безусловно, интересный пример схемы, которая, по всей видимости, «помнит», какой из двух входов последним был под напряжением.

Тем не менее намного более полезным является то, что эта схема запоминает, был ли определенный сигнал равен 0 или 1 *в конкретный момент времени*.

Давайте подумаем, как должна работать такая схема, прежде чем приступить к ее конструированию. Она будет иметь два входа. Назовем один из них «Данные». Как и все цифровые сигналы, вход «Данные» может иметь значение 0 или 1. Второй вход назовем «Запомнить этот бит», что является цифровым эквивалентом призыва «Запомните эту мысль». Обычно значение сигнала «Запомнить этот бит» равно 0, в случае чего сигнал «Данные» не влияет на схему. Когда значение сигнала «Запомнить этот бит» равно 1, выход схемы совпадает со значением сигнала «Данные». Затем сигнал «Запомнить этот бит» может вернуться к значению 0, и в это время схема запоминает последнее значение сигнала «Данные». Любые дальнейшие изменения в сигнале «Данные» не влияют на схему.

Другими словами, нам нужна схема со следующей функциональной таблицей.

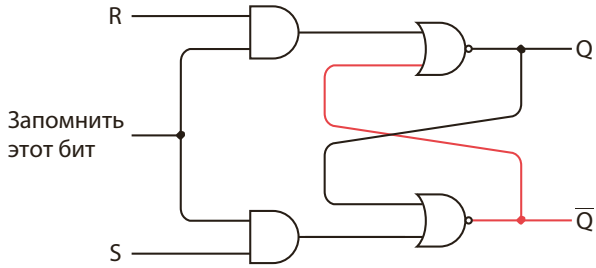
Входы		**Выход**
Данные	**Запомнить этот бит**	**Q**
0	1	0
1	1	1
0	0	Q
1	0	Q

В первых двух случаях, когда сигнал «Запомнить этот бит» равен 1, выход Q имеет то же значение, что и вход «Данные». В остальных двух случаях, когда сигнал «Запомнить этот бит» равен 0, значение выхода Q остается прежним. Обратите внимание: в этих двух случаях, когда сигнал «Запомнить этот бит» равен 0, выход Q является тем же самым, независимо от значения входа «Данные». Эту функциональную таблицу можно упростить.

Входы		**Выход**
Данные	**Запомнить этот бит**	**Q**
0	1	0
1	1	1
X	0	Q

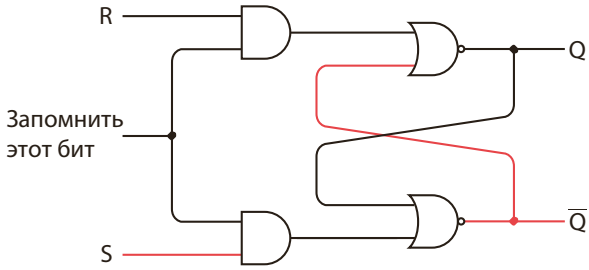
X означает «неважно». Значение входа «Данные» неважно, поскольку в случае, когда значение входа «Запомнить этот бит» равно 0, выход Q остается прежним.

Реализация сигнала «Запомнить этот бит» на основе существующего RS-триггера требует добавления на вход двух вентилей И.

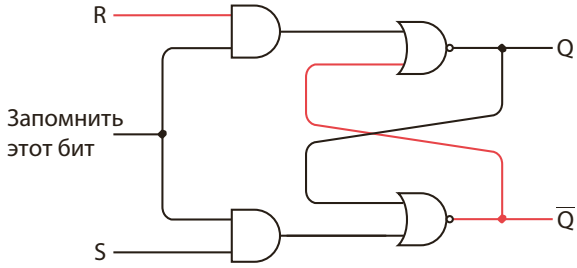


Напомним, что выход вентиля И равен 1, только если оба входа равны 1. На этой диаграмме выход Q — 0, а выход $\overline{Q}$ — 1.

Пока сигнал «Запомнить этот бит» равен 0, сигнал S не влияет на значения выходов.



Не влияет и сигнал R.



Только в случае, когда сигнал «Запомнить этот бит» равен 1, эта схема будет работать так же, как показанный ранее обычный RS-триггер.

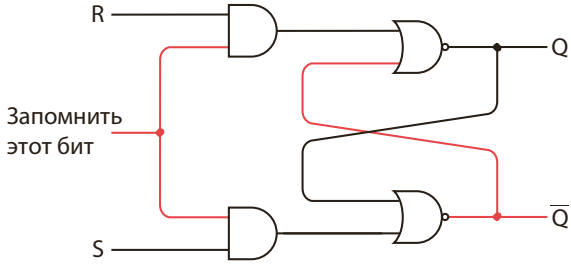
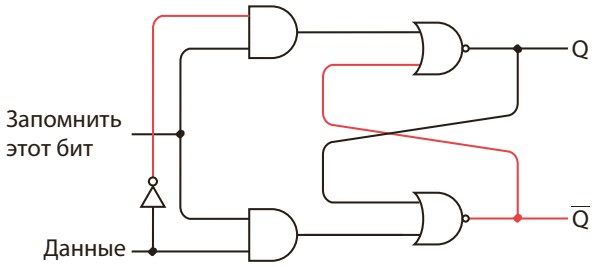


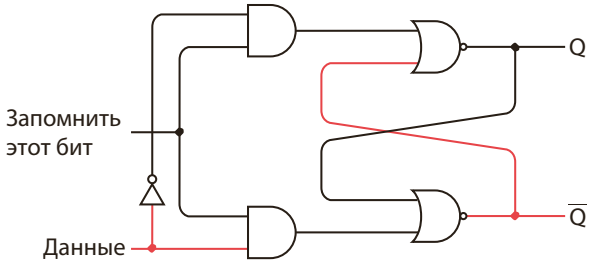
Схема ведет себя как обычный RS-триггер, поскольку теперь выход верхнего вентиля И совпадает с сигналом R, а выход нижнего вентиля И — с сигналом S.

Однако мы еще не достигли своей цели. Нам нужны только два входа, а не три. Как их уменьшить? Согласно исходной функциональной таблице RS-триггера, случай, когда сигналы S и R равны 1, запрещен, поэтому нужно его избежать. Кроме того, не имеет смысла и равенство этих сигналов 0, поскольку это просто говорит о неизменности выходного сигнала. В случае с этой схемой мы можем добиться того же результата, установив значение сигнала «Запомнить этот бит» равным 0.

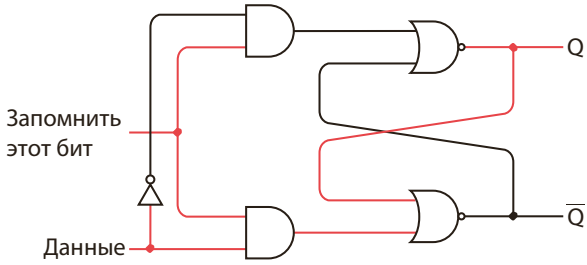
Имеет смысл, чтобы при значении сигнала S, равном 1, сигнал R становился равным 0, а при значении сигнала S, равном 0, сигнал R был равен 1. Сигнал под названием «Данные» может быть эквивалентен сигналу S, а инвертированный сигнал «Данные» — соответствовать сигналу R.



В данном случае оба входа равны 0, и выход Q равен 0 (выход $\overline{Q}$ — 1). Пока сигнал «Запомнить этот бит» равен 0, вход «Данные» не влияет на схему.



Когда сигнал «Запомнить этот бит» — 1, выход схемы равен значению входа «Данные».



Сейчас значение выхода Q равно значению входа «Данные», а значение выхода $\overline{Q}$ противоположно ему. Теперь сигнал «Запомнить этот бит» может вернуться к значению 0.

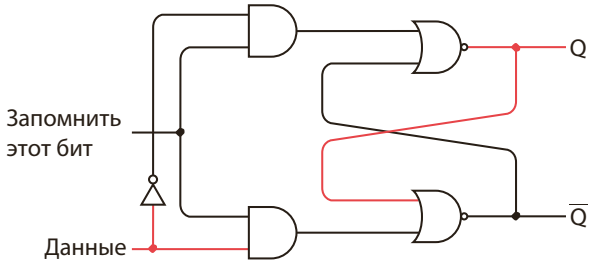
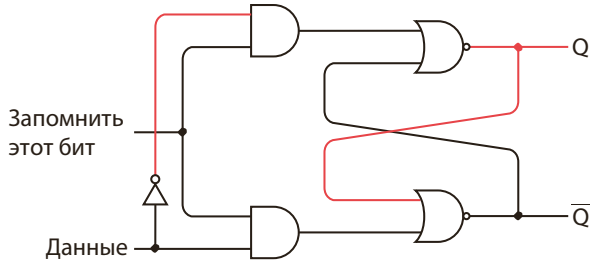


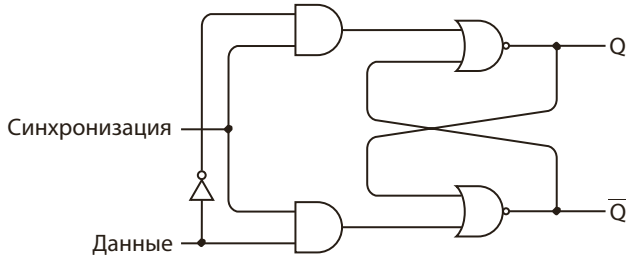
Схема запомнила значение сигнала «Данные» в момент, когда значение сигнала «Запомнить этот бит» последний раз было равно 1, независимо от изменения сигнала «Данные». Например, сигнал «Данные» мог бы вернуться к значению 0, не повлияв на выход.



Такая схема называется *D-триггером со срабатыванием по уровню*. Буква *D* означает «данные» (Data). *Срабатывание по уровню* указывает на то, что триггер сохраняет значение входа «Данные» в тот момент, когда сигнал на входе «Запомнить этот бит» достигает определенного *уровня*, в данном случае 1. (Далее мы рассмотрим альтернативу триггерам со срабатыванием по уровню.)

Обычно, когда такая схема фигурирует в специальной литературе, для входа используется обозначение не «Запомнить этот бит», а «Синхронизация»

(Clock). Иногда этот сигнал может обладать свойствами метронома, который с определенной регулярностью колеблется между значениями 0 и 1. Однако в нашем случае вход «Синхронизация» просто определяет момент, когда необходимо сохранить входной сигнал «Данные».



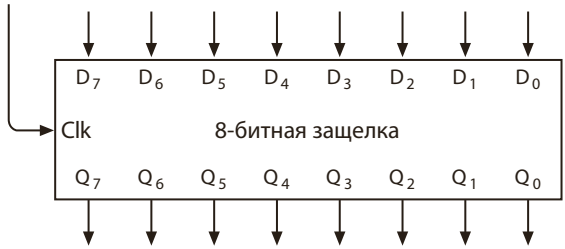
Как правило, в функциональной таблице вход «Данные» (Data) обозначается буквой *D*, а вход «Синхронизация» (Clock) — буквами *Clk*.

Входы		**Выходы**	
D	**Clk**	**Q**	**Q**
0	1	0	1
1	1	1	0
X	0	Q	Q

Эта схема также называется *защелкой* D-типа со срабатыванием по уровню; термин означает, что схема «запирает» один бит данных и удерживает его для дальнейшего использования. Эту схему также можно рассматривать в качестве *ячейки памяти* емкостью один бит. В главе 16 я продемонстрирую способ соединения большого количества таких триггеров для обеспечения памяти большего объема.

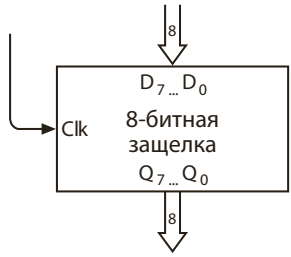
Часто в защелках полезно сохранять многобитное значение. Предположим, вы хотите использовать сумматор из главы 12 для сложения трех 8-битных чисел. Как обычно, вы вводите первое число с помощью первого набора переключателей, а второе — с помощью второго набора, затем потребуется записать результат на бумаге. После этого нужно будет ввести этот результат с помощью одного набора переключателей, а с помощью второго — ввести третье число. На самом деле вводить промежуточный результат нет необходимости. Вы можете использовать его, взяв непосредственно из первого расчета.

Давайте решим эту задачу, применяя защелки. Объединим восемь защелок, каждая из которых использует два вентиля ИЛИ-НЕ, два вентиля И и один инвертор, как было показано ранее. Все входы «Синхронизация» соединены между собой. Вот что у нас получилось.

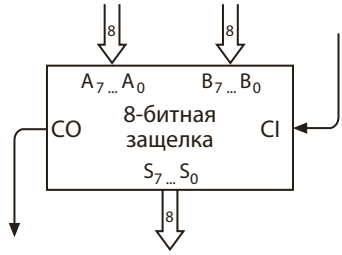


Эта защелка способна одновременно хранить восемь бит информации. Восемь входов сверху обозначены от D_0 до D_7, а восемь выходов внизу — от Q_0 до Q_7. Слева расположен вход «Синхронизация» (Clk). Сигнал Clk обычно равен 0. Когда сигнал Clk — 1, то 8-битное значение на входах D передается на выходы Q. Когда сигнал Clk возвращается к 0, тогда 8-битное значение сохраняется там, пока сигнал Clk снова не станет равен 1.

Восьмибитную защелку также можно изобразить с восемью входами «Данные» и восемью выходами Q, сгруппированными вместе.



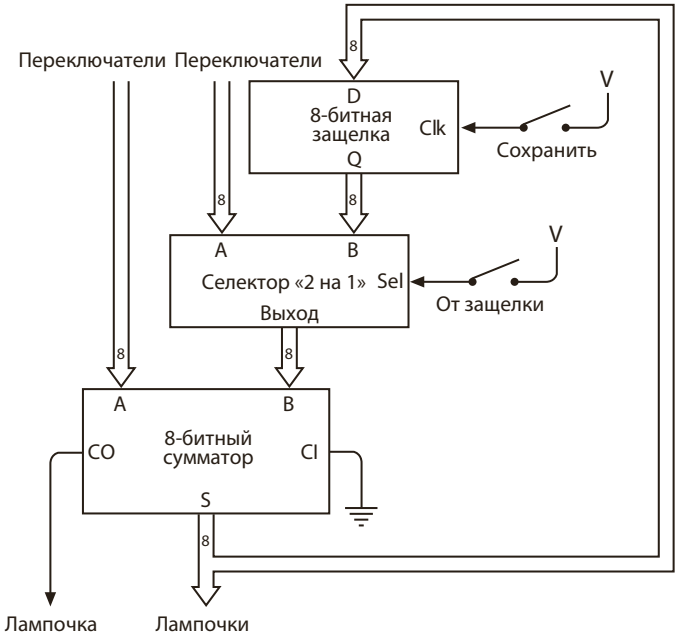
Вот схема 8-битного сумматора.



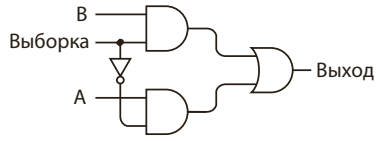
Обычно (если проигнорировать то, что мы делали с вычитанием в предыдущей главе), восемь входов A и восемь входов B подключены к переключателям, вход для переноса CI — к земле, а восемь выходов S и CO — к лампочкам.

В этой пересмотренной версии восемь выходов S 8-разрядного сумматора могут быть подключены как к лампочкам, *так и* ко входам D 8-битной защелки.

Для сохранения результата работы сумматора можно подключить переключатель «Сохранить» ко входу Clk защелки.



Селектор двух линий на одну позволяет выбрать с помощью переключателя, откуда должны поступать данные на входы B: из второго ряда переключателей или из выходов Q защелки. Замыкание переключателя означает выбор выходов 8-битной защелки. В селекторе «2 на 1» используется восемь следующих схем.

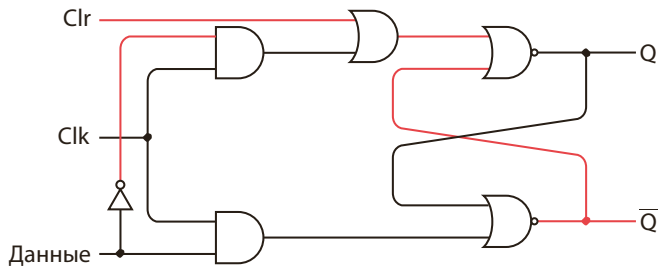


Если вход «Выборка» (или Sel — сокращение от английского Select) равен 1, то значение сигнала на выходе вентиля ИЛИ равно значению сигнала на входе B. Это связано с тем, что выход верхнего вентиля И равен входу B, а выход нижнего вентиля И — 0. Если вход «Выборка» равен 0, выход будет совпадать со входом A. Эти правила представлены в следующей функциональной таблице.

Входы			Выходы
Выборка	**A**	**B**	**Q**
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

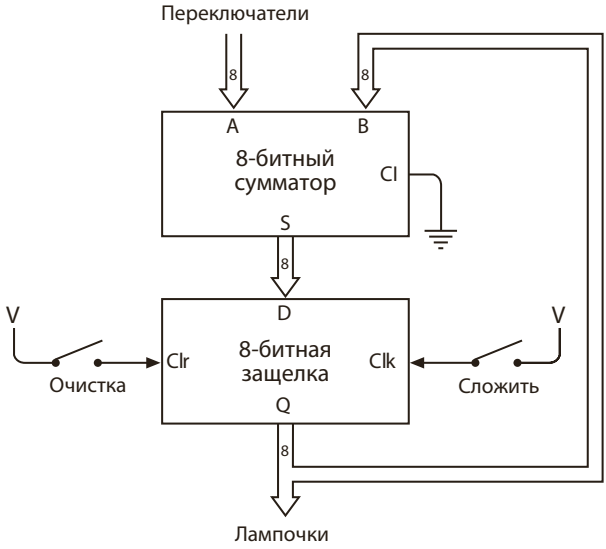
Селектор, являющийся частью модифицированного сумматора, включает восемь таких однобитных селекторов. Все входы «Выборка» соединены между собой.

Этот модифицированный сумматор не вполне корректно обрабатывает сигнал CO (выход для переноса). Если при сложении двух чисел этот сигнал CO становится равным 1, то этот сигнал игнорируется при добавлении к сумме следующего числа. Одно из возможных решений заключается в том, чтобы создать сумматор, защелку и селектор, разрядность которых составляет 16 бит или, по крайней мере, превышает разрядность наибольшей суммы, которая может получиться. Мы не будем приступать к решению этой задачи вплоть до главы 17. Более интересный подход к созданию сумматора позволяет обойтись без ряда из восьми переключателей. Однако сначала нужно немного изменить триггер типа D, добавив вентиль ИЛИ и входной сигнал «Очистка» (Clear, или Clr). Вход Clr обычно равен 0. Когда он равен 1, выход Q — 0.



Сигнал Q становится равным 0 вне зависимости от других входных сигналов, что приводит к стиранию информации, сохраненной в триггере.

Вы можете спросить: зачем это нужно? Почему мы не можем очистить триггер, подав на вход «Данные» 0, а на вход Clk — 1? Может быть, мы не можем точно контролировать то, что подается на вход «Данные»? Возможно, у нас есть набор из восьми таких защелок, подключенных к выходам 8-битного сумматора, как показано ниже.

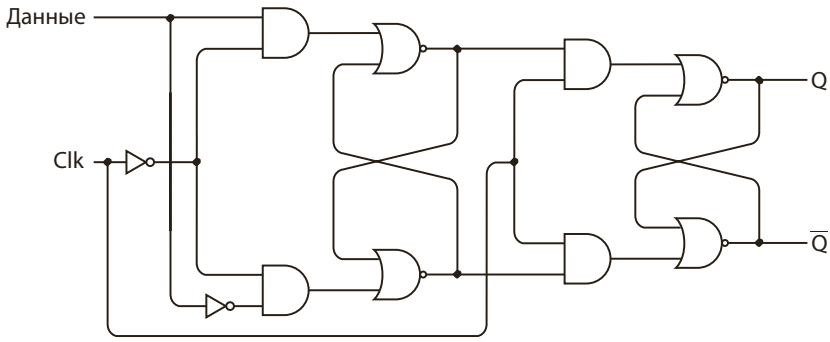


Обратите внимание: переключатель с меткой «Сложить» теперь управляет входом защелки Clk.

Может показаться, что этот сумматор использовать проще, чем предыдущий, особенно если требуется сложить множество чисел. Сначала вы нажимаете кнопку «Очистка». В результате выходы защелок становятся равными 0, все лампочки отключаются, а на второй набор входов 8-битного сумматора подаются значения 0. Вы вводите первое число и нажимаете кнопку «Сложить». Это число отображается в виде комбинации горящих лампочек. Затем вводите второе число и снова нажимаете кнопку «Сложить». Число, введенное с помощью переключателей, добавляется к предыдущей сумме, и результат снова отображается с помощью лампочек. Вы можете продолжать вводить новые числа и нажимать кнопку «Сложить». Я уже говорил, что сконструированный нами D-триггер *срабатывает по уровню*, то есть *уровень* сигнала на входе Clk должен измениться с 0 на 1, чтобы в защелке сохранилось значение на входе «Данные». Пока сигнал на входе Clk равен 1, значение входа «Данные» может меняться; любые изменения входа «Данные», пока сигнал Clk равен 1, будут отражаться в значениях выходов Q и $\overline{Q}$.

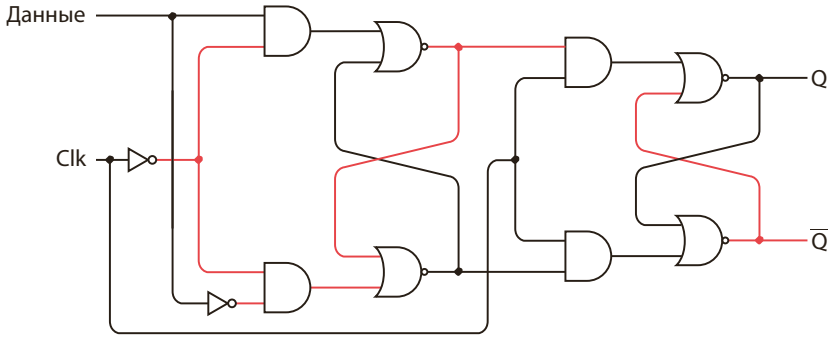
Для решения некоторых задач бывает достаточно входа Clk со срабатыванием по уровню. Для решения других более предпочтительным является вход Clk *со срабатыванием по фронту*. В этом случае выходы изменяются только *во время перехода значения сигнала Clk от 0 к 1*. Как и при использовании триггера со срабатыванием по уровню, когда вход Clk равен 0, любые изменения входного сигнала «Данные» не влияют на выходы. Отличие

триггера со срабатыванием по фронту заключаются в том, что изменения входного сигнала «Данные» не воздействуют на выходы даже тогда, когда вход Clk равен 1. Входной сигнал «Данные» влияет на выходы только в тот момент, когда значение входного сигнала Clk меняется с 0 на 1. D-триггер со срабатыванием по фронту состоит из двух блоков RS-триггера, соединенных следующим образом.

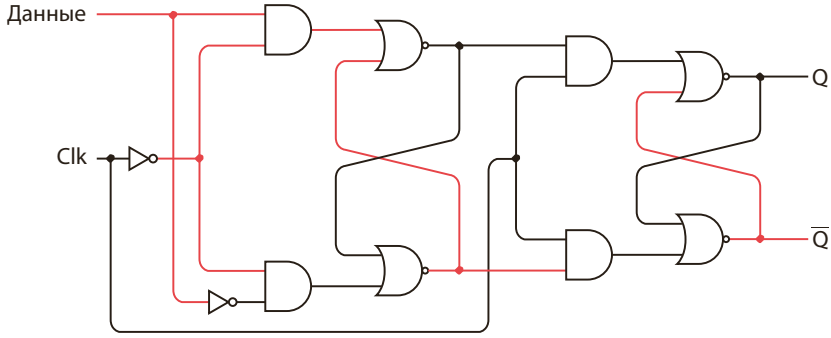


В данном случае идея в том, что вход Clk управляет как первым, так и вторым блоком. Однако в первом блоке сигнал Clk инвертируется, или первый блок работает так же, как D-триггер, за исключением того, что входной сигнал «Данные» сохраняется, когда сигнал Clk равен 0. Выходы второго блока — входы первого, и их сигналы сохраняются, когда вход Clk равен 1. В итоге входной сигнал «Данные» сохраняется в момент изменения сигнала Clk с 0 на 1.

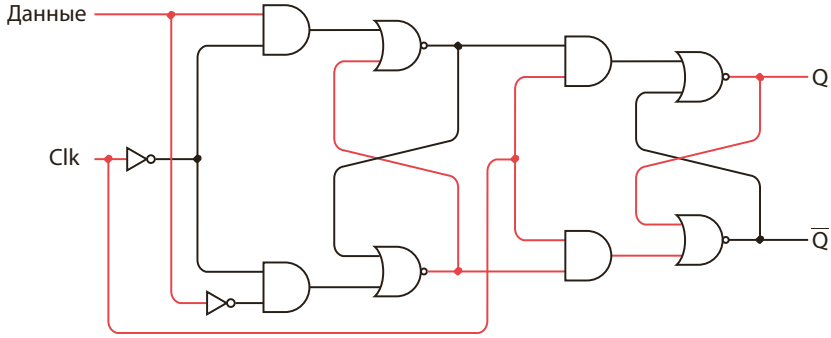
Давайте рассмотрим эту схему подробно. На следующем изображении показан триггер в состоянии покоя, когда входы «Данные» и Clk, а также выход Q равны 0.



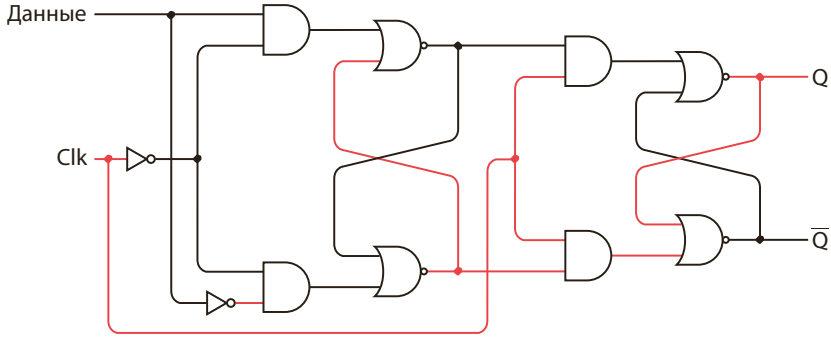
Теперь измените значение входного сигнала «Данные» на 1.



Это изменит состояние первого триггера, поскольку инвертированный входной сигнал Clk равен 1. Однако второй блок остается неизменным, так как неинвертированный входной сигнал Clk равен 0. Теперь измените входной сигнал Clk на 1.



Это приведет к изменению второго блока, при этом значение выхода Q поменяется на 1. Разница заключается в том, что входной сигнал «Данные» теперь может меняться (например, обратно на 0), не влияя на значение выхода Q.

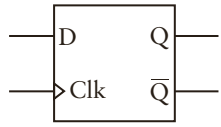


Выходы Q и $\overline{Q}$ могут меняться только в тот момент, когда входной сигнал Clk изменяется с 0 на 1.

В функциональную таблицу D-триггера со срабатыванием по фронту требуется добавить новый символ — стрелку вверх (↑), которая обозначает изменение значения сигнала с 0 на 1.

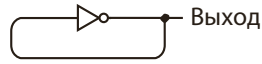
Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
0	↑	0	1
1	↑	1	0
X	0	Q	Q

Стрелка указывает на то, что значение выходного сигнала Q становится равным входному сигналу «Данные» в момент изменения значения сигнала Clk с 0 на 1. Этот процесс называется *положительным переходом* сигнала Clk (*отрицательный* переход с 1 к 0). Схема триггера выглядит следующим образом.

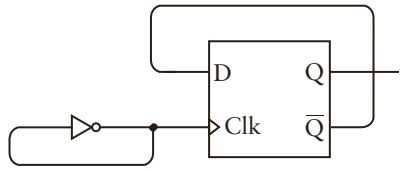


Маленькая угловая скобка указывает на то, что триггер срабатывает по фронту.

Теперь хочу показать вам схему, использующую D-триггер, срабатывающий по фронту, которую нельзя продублировать с помощью триггера, срабатывающего по уровню. Вспомните осциллятор, который мы создали в начале этой главы. Значение выходного сигнала этого осциллятора чередуется между 0 и 1.



Давайте подключим выход осциллятора ко входу Clk D-триггера, срабатывающего по фронту, а выход $\overline{Q}$ — ко входу D.



Выход триггера одновременно является его же входом. Это обратная связь с обратной связью! На практике это может привести к проблеме. Осциллятор

сконструирован из реле, которое вибрирует с максимально возможной скоростью. Выход осциллятора подключен к реле, из которых сконструирован триггер. Эти другие реле могут не поспевать за скоростью осциллятора. Чтобы избежать этого, предположим, что реле, используемое в осцилляторе, работает медленнее, чем реле, применяемые в других местах этой схемы.

Чтобы понять, что происходит в этой схеме, давайте посмотрим на функциональную таблицу, иллюстрирующую различные изменения. Допустим, что вход Clk и выход Q равны 0. Значит, выход Q, подключенный к входу D, равен 1.

Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
1	0	0	1

Когда значение входного сигнала Clk изменяется с 0 на 1, выходной сигнал Q становится равным входному сигналу D.

Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
0	↑	0	1
1	↑	1	0

Поскольку выходной сигнал $\overline{Q}$ изменяется на 0, входной сигнал D также поменяется на 0. Теперь входной сигнал Clk станет равен 1.

Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
1	0	0	1
1	↑	1	0
0	1	1	0

Входной сигнал Clk возвращается к 0, не влияя на значения выходных сигналов.

Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0

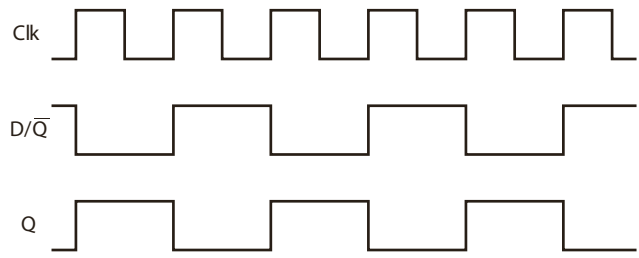
Теперь значение входного сигнала Clk снова изменяется на 1. Поскольку входной сигнал D равен 0, выходной сигнал Q становится равным 0, а выходной сигнал $\overline{Q}$ — 1.

Входы		Выходы	
D	**Clk**	**Q**	**$\overline{Q}$**
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

Таким образом, входной сигнал D также становится равным 1.

Входы		Выходы	
D	**Clk**	**Q**	**$\overline{Q}$**
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1

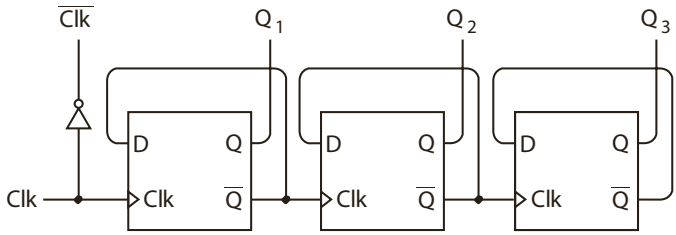
То, что здесь происходит, можно описать очень просто: каждый раз, когда значение входного сигнала изменяется с 0 на 1, значение выходного сигнала Q меняется либо с 0 на 1, либо с 1 на 0. Ситуацию может прояснить следующий график.



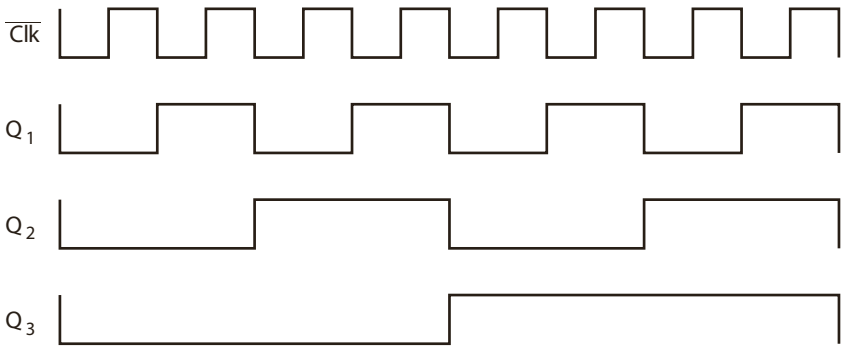
Когда входной сигнал Clk изменяется с 0 на 1, значение на входе D (которое совпадает со значением на выходе Q) передается на выход Q, при этом также изменяется значение Q и D до следующего перехода значения входного сигнала Clk от 0 к 1.

Если частота осциллятора равна 20 герц (20 циклов в секунду), частота выхода Q в два раза меньше — 10 герц. По этой причине такая схема, в которой выход Q соединен со входом триггера «Данные», также называется *делителем частоты*.

Разумеется, выход делителя частоты может являться входом Clk другого делителя частоты для повторного деления частоты. На следующем изображении показана схема, состоящая из трех делителей частоты.

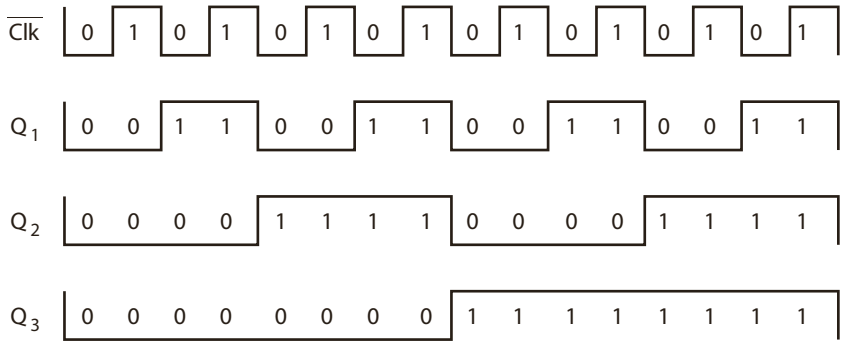


Давайте рассмотрим четыре сигнала, которые я обозначил в верхней части схемы.



Признаю, что я выбрал самые подходящие точки начала и окончания этой диаграммы, однако ничего нечестного в этом нет: этот шаблон повторяется в данной схеме снова и снова. Не кажется ли она знакомой?

Подскажу: обозначим эти сигналы нулями и единицами.



Не догадались? Попробуйте повернуть диаграмму на 90 градусов по часовой стрелке и прочитать 4-битные числа по горизонтали. Каждое из них соответствует десятичному числу от 0 до 15.

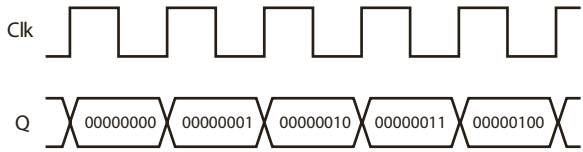
Двоичное число	**Десятичное число**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Эта схема *производит подсчет в двоичном формате*, и чем больше триггеров добавляем в схему, тем большую сумму можно получить с ее помощью. В главе 8 я указал, что в последовательности возрастающих двоичных чисел каждый столбец цифр чередуется между 0 и 1 с частотой вдвое меньшей, чем столбец справа от него. Этот счетчик имитирует эту закономерность. При каждом положительном переходе сигнала Clk значения выходных сигналов счетчика увеличиваются на 1, это и называется *приращением* или *инкрементом*.

Давайте объединим восемь триггеров и поместим их в общий корпус.

Схема называется *сквозным счетчиком*, потому что выход каждого триггера становится входом Clk следующего. Изменения сигнала проходят последовательно через все триггеры, а изменения триггеров, находящихся в конце, могут происходить с небольшой задержкой. Более сложные счетчики — *синхронные*, в них все выходные сигналы изменяются одновременно.

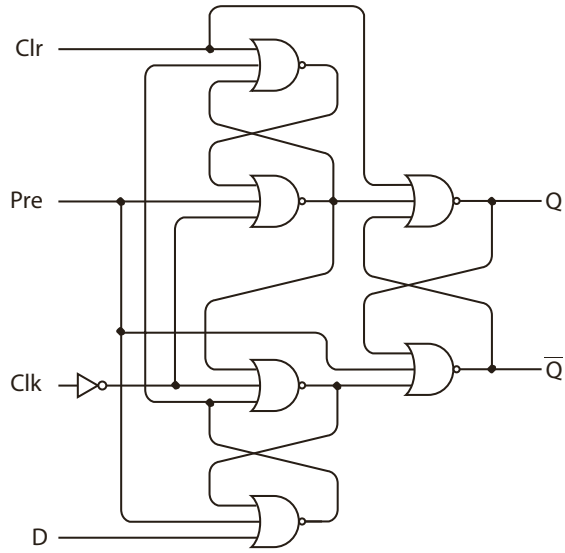
Я обозначил выходы буквами от Q_0 до Q_7. Они расположены так, что выход первого триггера в цепочке (Q_0) является крайним справа. Если вы подключите к этим выходам лампочки, сможете прочитать 8-битное число. Временна́я диаграмма такого счетчика может отображать все восемь выходов отдельно или вместе следующим образом.



При каждом положительном переходе сигнала Clk некоторые выходы Q могут измениться, некоторые — нет, однако вместе они отражают последовательность возрастающих двоичных чисел.

Ранее в этой главе я говорил, что мы найдем способ определения частоты осциллятора. Итак, если вы подключите осциллятор к входу Clk 8-битного счетчика, последний покажет, сколько колебаний совершил осциллятор. Когда общее число достигнет 11111111 (255 в десятичной системе счисления), счетчик вернется к 00000000. Вероятно, самый простой способ определения частоты осциллятора заключается в том, чтобы подключить восемь лампочек к выходам 8-битного счетчика. Теперь подождите, пока значения всех выходов не станут равны 0, то есть пока не погаснут все лампочки, и запустите секундомер. Остановите секундомер, когда все лампочки погаснут снова. Это время, необходимое для 256 колебаний. Скажем, для этого требуется десять секунд. Таким образом, частота осциллятора составляет 256 / 10 = 25,6 герца.

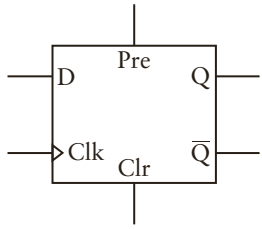
С появлением в триггерах дополнительных функций они становятся более сложными. На следующем рисунке изображен *D-триггер с предустановкой и очисткой, срабатывающий по фронту*.



Входы Pre (от «preset» — «предустановка») и Clr обладают более высоким приоритетом, чем входы Clk и «Данные». Обычно эти два входа равны 0. Когда вход Pre равен 1, выход Q становится равным 1, а выход $\overline{Q}$ — 0. Когда вход Clr равен 1, выход Q — 0, а выход $\overline{Q}$ — 1. Как и в случае со входами S и R RS-триггера, входы Pre и Clr не должны одновременно быть равными 1. В остальном этот триггер ведет себя как обычный D-триггер со срабатыванием по фронту.

Входы				**Выходы**	
Pre	**Clr**	**D**	**Clk**	**Q**	**$\overline{Q}$**
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	Q

D-триггер с предустановкой и очисткой, срабатывающий по фронту, обозначается следующим образом.



Теперь наши телеграфные реле умеют складывать, вычитать и производить подсчет в двоичном формате. Это значительное достижение, особенно учитывая то, что используемое оборудование было доступно более ста лет назад. Впереди у нас еще множество открытий. Однако давайте немного отдохнем от конструирования схем и вернемся к системам счисления.

Помимо упомянутых RS- и D-триггеров, существуют еще два основных вида: JK-триггеры (Jerk/Kill, внезапное включение / внезапное отключение) и T-триггеры (Toggle, переключатель; он же счетный триггер). Таблица истинности JK-триггера отличается от таковой для RS-триггера наличием перехода при J = K = 1, где ввод J аналогичен вводу S, а ввод K — вводу R. Таким образом, это отличие можно записать следующим образом.

Входы			**Выходы**	
Q(t – 1)	**J**	**K**	**Q(t)**	**Q(t)**
0	1	1	1	0
1	1	1	0	1

Получается, J = K = 1 инвертирует предыдущее состояние триггера. Во всем остальном принцип работы аналогичен RS-триггеру.

Принцип действия T-триггера заключается в том, что он инвертирует входной сигнал.

Входы		**Выходы**	
Q(t – 1)	**T**	**Q(t)**	**Q(t)**
0	0	0	1
1	0	1	0
0	1	1	0
1	1	0	1

Глава 15
Байты и шестнадцатеричные числа

Две усовершенствованные счетные машины, описанные в предыдущей главе, хорошо иллюстрируют концепцию *потоков данных*. Восьмибитные значения перемещаются по цепи от одного компонента к другому. Эти значения подаются на входы сумматоров, защелок и селекторов, а также появляются на выходах этих устройств. Кроме того, 8-битные значения задаются с помощью переключателей и отображаются рядом лампочек. Таким образом, поток данных в этих схемах имеет *ширину восемь бит*. Но почему? Почему не шесть, не семь, не девять и не десять?

Можно было бы ответить, что основой наших усовершенствованных счетных машин является исходный сумматор из главы 12, работающий с 8-битными значениями. Однако нет никаких особых причин конструировать эту машину именно так. Просто при ее создании мы сочли эту величину удобной. Как бы то ни было, признаю, что схитрил, поскольку с самого начала знал (возможно, и вы тоже), что восемь бит данных соответствуют одному байту.

Слово «байт» (byte) возникло в компании IBM примерно в 1956 году. Оно произошло от слова bite («кусок»), но его было решено писать через букву *y*, чтобы не путать со словом bit («бит»). В течение некоторого времени слово «байт» обозначало просто число битов в конкретном потоке данных. Однако в середине 1960-х, в связи с разработкой семейства компьютеров System/360 в компании IBM, это слово стало обозначать группу из восьми бит.

Как 8-разрядное число, байт может принимать значения в диапазоне от 00000000 до 11111111. Эти значения могут описывать положительные целые числа от 0 до 255, а при использовании дополнения до двойки для представления отрицательных чисел они могут отображать как положительные, так и отрицательные целые числа в диапазоне от −128 до 127. Кроме того, конкретный байт может просто представлять одну из 2^8, или 256, разных вещей.

Число 8 оказалось весьма удобной величиной. Компания IBM отдала предпочтение 8-битным байтам в связи с простотой хранения чисел в формате BCD (о котором я расскажу в главе 23). Однако, как мы увидим далее, байт идеально подходит для хранения текста, поскольку бóльшую часть языков мира (за исключением идеограмм, использующихся в китайском, японском и корейском) можно представить менее чем 256 символами. Кроме того, байт идеально подходит для представления оттенков серого на черно-белых фотографиях, поскольку человеческий глаз различает примерно 256 оттенков этого цвета. А там, где не хватает одного байта (например, для представления вышеупомянутых идеограмм китайского, японского и корейского языков), как правило, можно использовать два байта, которые позволяют выразить 2^{16}, или 65 536, различных элементов.

Половина байта, то есть четыре бита, иногда называется *тетрадой*, однако это слово употребляется гораздо реже, чем «байт».

Поскольку байты часто используются при описании работы компьютеров, нам требуется как можно более лаконичный способ записи их значения. Например, запись числа, состоящего из восьми двоичных цифр 10110110, безусловно, является корректной, но едва ли лаконичной.

Разумеется, мы всегда можем обращаться к байтам, используя их десятичные эквиваленты, но это потребует преобразования двоичного числа в десятичное, что хоть и несложно, но весьма обременительно. В главе 8 я продемонстрировал один довольно простой подход. Поскольку каждая двоичная цифра соответствует степени 2, мы можем просто записать цифры двоичного числа, а под ними — степени 2, после чего перемножить числа в каждом столбце и сложить результаты. Далее представлен процесс преобразования числа 10110110.

1	0	1	1	0	1	1	0
×128	×64	×32	×16	×8	×4	×2	×1

128 + 0 + 32 + 16 + 0 + 4 + 2 + 0 = 182

Процесс преобразования десятичного числа в двоичное менее удобен и предполагает деление десятичного числа на убывающие степени двойки. Частное от каждого деления — двоичная цифра, а остаток делится на следующую в порядке убывания степень двойки. Вот как десятичное число 182 преобразуется в двоичное.

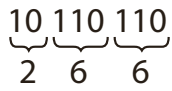
182	54	54	22	6	6	2	0
:128	:64	:32	:16	:8	:4	:2	:1
1	0	1	1	0	1	1	0

В главе 8 эта техника описана подробно. Тем не менее для преобразования двоичных чисел в десятичные и обратно обычно требуется бумага, карандаш и практика.

Мы уже узнали о восьмеричной системе счисления — системе счисления с основанием 8, где используются только цифры 0, 1, 2, 3, 4, 5, 6 и 7. Преобразовать восьмеричное число в двоичное легко. Все, что нужно, — это запомнить 3-битный эквивалент каждой восьмеричной цифры.

Двоичное число	Восьмеричное число
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Если у вас есть двоичное число (например, 10110110), начинайте преобразование с правого края. Каждая группа из трех бит соответствует восьмеричной цифре.



Таким образом, байт 10110110 можно выразить в виде восьмеричного числа 266. Это выражение, безусловно, является более лаконичным, значит, восьмеричная система действительно подходит для представления байтов. Однако у нее есть небольшой недостаток.

В двоичной системе байты выражаются значениями в диапазоне от 00000000 до 11111111, в восьмеричной — значениями в диапазоне от 000 до 377. Как было показано в предыдущем примере, средней и крайней справа восьмеричным цифрам соответствуют группы из трех бит, однако крайней слева восьмеричной цифре соответствуют только два бита. Это означает, что восьмеричное представление 16-разрядного числа не совпадает с восьмеричными представлениями двух байтов, составляющих это 16-разрядное число.

Чтобы согласовать представления многобайтных значений с представлениями отдельных байтов, нужна система, в которой каждый байт делится на равное количество битов. Следовательно, нам требуется разделить каждый байт на четыре значения по два бита каждое (система счисления с основанием 4) или на два значения по четыре бита каждое (система счисления с основанием 16).

Систему счисления с основанием 16 мы еще не рассматривали, и на то есть причины. Система счисления с основанием 16 называется *шестнадцатеричной**, — даже название труднопроизносимо. В десятичной системе счисления считаем так:

0 1 2 3 4 5 6 7 8 9 10 11 12 …

В восьмеричной системе, как вы помните, не используются цифры 8 и 9:

0 1 2 3 4 5 6 7 10 11 12 …

В системе с основанием 4 не требуются цифры 4, 5, 6 и 7:

0 1 2 3 10 11 12 …

Наконец, в двоичной системе достаточно только 0 и 1:

0 1 10 11 100 …

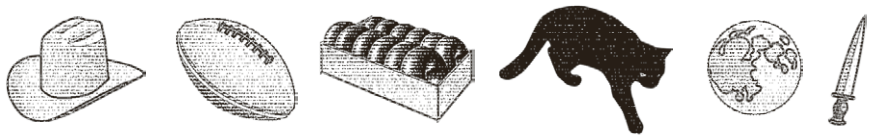
Однако шестнадцатеричная система отличается тем, что в ней используется *больше* цифр, чем в десятичной. В шестнадцатеричной системе подсчет происходит примерно так:

0 1 2 3 4 5 6 7 8 9 ? ? ? ? ? ? 10 11 12 …

В данном случае 10 соответствует числу $16_{\text{десять}}$. Вопросительные знаки говорят о том, что нам нужны еще шесть символов для представления шестнадцатеричных чисел. Что это за символы? Откуда их брать? Что ж, поскольку

* Шестнадцатеричная система счисления используется для обозначения MAC-адресов (уникальных физических адресов сетевого оборудования) и для записи сетевых (IP) адресов в современном протоколе IPv6. С этим вы можете столкнуться при настройке доступа в интернет на своем компьютере. *Прим. науч. ред.*

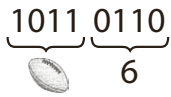
они не достались нам в наследство, подобно другим традиционным числовым символам, мы можем придумать их самостоятельно, например такие.



В отличие от символов, используемых для обозначения большинства чисел, у этих обозначений есть преимущество: они легко запоминаются и отождествляются с теми величинами, которые представляют. Существует так называемая десятигаллонная ковбойская шляпа, мяч для американского футбола (11 игроков в команде), дюжина пончиков (12 штук), черная кошка (с которой ассоциируется несчастливое число 13), полная луна (появляется на небе через 14 дней после новолуния) и кинжал (напоминающий об убийстве Юлия Цезаря в 15-й день марта). Каждый байт можно выразить в виде двух шестнадцатеричных цифр. Другими словами, шестнадцатеричная цифра эквивалентна четырем битам, или одной тетраде. В следующей таблице показаны соответствия двоичных, шестнадцатеричных и десятичных чисел.

Двоичное число	**Шестнадцатеричное число**	**Десятичное число**	**Двоичное число**	**Шестнадцатеричное число**	**Десятичное число**
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010		10
0011	3	3	1011		11
0100	4	4	1100		12
0101	5	5	1101		13
0110	6	6	1110		14
0111	7	7	1111		15

Вот как можно представить двоичное число 10110110 в шестнадцатеричной системе.



И не важно, имеем ли мы дело с многобайтными числами.

Один байт всегда представляется парой шестнадцатеричных цифр.

К сожалению (а может быть, к счастью), мы не собираемся использовать футбольные мячи и пончики для записи шестнадцатеричных чисел, хотя они, безусловно, могли бы сгодиться для этой цели. Вместо них в шестнадцатеричной системе применяются обозначения, приводящие многих в замешательство. Дело в том, что шесть недостающих шестнадцатеричных цифр представляют шестью первыми буквами латинского алфавита:

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 …

В следующей таблице показано *реальное* соответствие между двоичными, шестнадцатеричными и десятичными числами.

Двоичное число	**Шестнадцатеричное число**	**Десятичное число**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Таким образом, двоичное число 10110110 можно представить шестнадцатеричным числом B6, не рисуя футбольный мяч. Как вы помните, в предыдущих главах я указывал основание системы счисления с помощью нижнего индекса, например: $10110110_{\text{ДВА}}$ — для двоичной системы; $2312_{\text{ЧЕТЫРЕ}}$ — для четвертичной; $266_{\text{ВОСЕМЬ}}$ — для восьмеричной; $182_{\text{ДЕСЯТЬ}}$ — для десятичной.

По аналогии мы можем использовать обозначение $B6_{\text{ШЕСТНАДЦАТЬ}}$ для шестнадцатеричной системы.

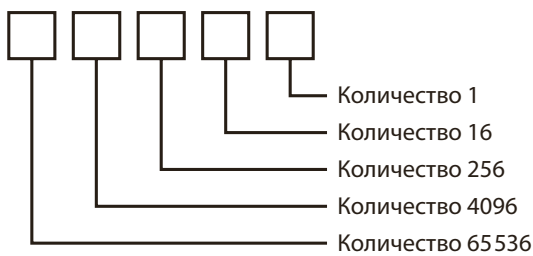
Однако такое выражение чересчур громоздко. К счастью, для шестнадцатеричных чисел существуют и другие, более краткие, обозначения. Вы можете записать такое число следующим образом:

$$B6_{\text{HEX}}.$$

В этой книге я буду использовать распространенный способ представления шестнадцатеричных чисел, предполагающий добавление к числу строчной латинской буквы *h*:

B6h.

В шестнадцатеричном числе положение каждой цифры соответствует степени числа 16.



Шестнадцатеричное число 9A48Ch можно представить так:

$$\begin{aligned} 9A48Ch = {} & 9 \times 10000h + \\ & A \times 1000h + \\ & 4 \times 100h + \\ & 8 \times 10h + \\ & C \times 1h. \end{aligned}$$

Это выражение можно записать, используя степени числа 16:

$$\begin{aligned} 9A48Ch = {} & 9 \times 16^4 + \\ & A \times 16^3 + \\ & 4 \times 16^2 + \\ & 8 \times 16^1 + \\ & C \times 16^0. \end{aligned}$$

Или десятичные эквиваленты этих степеней:

$$\begin{aligned} 9A48Ch = {} & 9 \times 65\,536 + \\ & A \times 4096 + \\ & 4 \times 256 + \\ & 8 \times 16 + \\ & C \times 1. \end{aligned}$$

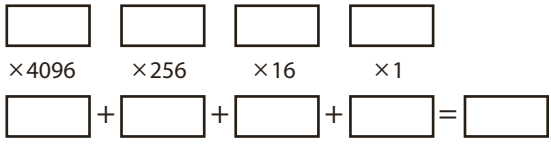
Обратите внимание на отсутствие двусмысленности при записи отдельных цифр числа (9, A, 4, 8 и C) без нижнего индекса, обозначающего основание системы счисления. Девять — это 9, будь то десятичная или шестнадцатеричная система счисления. С другой стороны, A очевидно представляет шестнадцатеричный эквивалент десятичного числа 10.

По сути, преобразование всех цифр в десятичные числа позволяет выполнить расчет итогового значения:

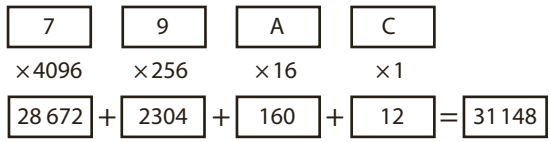
$$\begin{aligned} 9A48Ch = {} & 9 \times 65\,536 + \\ & 10 \times 4096 + \\ & 4 \times 256 + \\ & 8 \times 16 + \\ & 12 \times 1. \end{aligned}$$

В итоге получается число 631 948. Таким образом шестнадцатеричные числа преобразуются в десятичные.

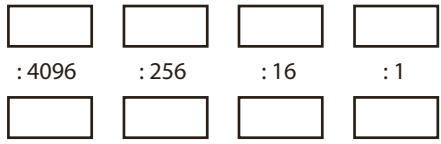
Шаблон для преобразования любого четырехзначного шестнадцатеричного числа в десятичное выглядит следующим образом.



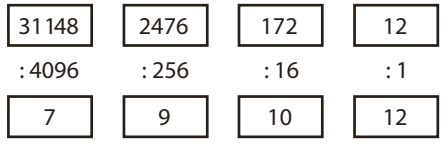
В качестве примера преобразуем число 79ACh. Имейте в виду, что шестнадцатеричные цифры A и C эквивалентны десятичным числам 10 и 12.



Преобразование десятичных чисел в шестнадцатеричные обычно предполагает выполнение операций деления. Число меньшее или равное 255 можно представить одним байтом, состоящим из двух шестнадцатеричных цифр. Чтобы вычислить эти две цифры, нужно разделить число на 16, в результате чего получится частное и остаток. Вернемся к примеру с десятичным числом 182. Разделив 182 на 16, получим 11 (что соответствует цифре B в шестнадцатеричной системе) и 6 в остатке. Так, шестнадцатеричным эквивалентом десятичного числа 182 является B6h. Если десятичное число, которое вы хотите преобразовать, меньше 65 536, то шестнадцатеричный эквивалент будет состоять не более чем из четырех цифр. Шаблон для преобразования такого числа в шестнадцатеричное следующий.



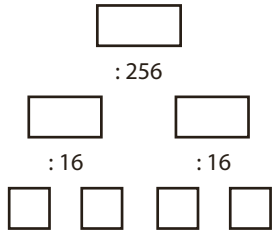
Сначала поместите десятичное число в верхний левый прямоугольник. Это наше первое делимое. Разделим число на 4096 (первый делитель). Частное впишем в прямоугольник, расположенный под делимым, а остаток — в прямоугольник справа от делимого. Этот остаток — новое делимое, которое мы разделим на 256. Вот как число 31 148 преобразуется в шестнадцатеричный формат.



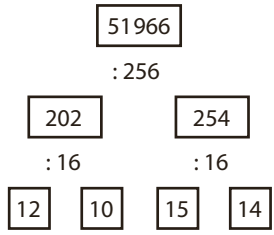
Десятичные числа 10 и 12 соответствуют шестнадцатеричным цифрам A и C, поэтому результат равен 79ACh.

Одна из проблем этой техники заключается в том, что для деления вы, вероятно, решите использовать калькулятор, а калькуляторы не показывают остаток от деления. Если вы разделите 31 148 на 4096 на калькуляторе, то получите 7,6044921875. Чтобы рассчитать остаток, нужно умножить 4096 на 7 (получится 28 672) и вычесть это значение из 31 148. Или умножить 4096 на 0,6044921875 — дробную часть результата от деления. (Правда, некоторые калькуляторы предусматривают функцию преобразования десятичных чисел в шестнадцатеричные и обратно.)

Другой способ преобразования десятичных чисел от 0 до 65 535 в шестнадцатеричные предполагает разделение числа на два байта путем его деления на 256. Затем каждый байт делится на 16. Шаблон для этого следующий.



Начнем сверху. После каждой операции деления частное помещается в прямоугольник, расположенный слева от делителя, а остаток — в прямоугольник справа. Например, число 51 966 преобразуется таким образом.



Шестнадцатеричными эквивалентами чисел 12, 10, 15 и 14 являются буквы *C, A, F и E*, поэтому результат скорее напоминает слово, чем число.

Далее представлена таблица сложения для шестнадцатеричной системы счисления.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Используя эту таблицу и обычные правила сложения в столбик, можно складывать шестнадцатеричные числа.

```
  4A3378E2
+ 877AB982
----------
  D1AE3264
```

В главе 13 я упоминал, что для представления отрицательных чисел можно применять дополнение до двойки. Если вы имеете дело с 8-битными двоичными числами со знаком, то отрицательные числа начинаются с 1. В шестнадцатеричной системе счисления двузначные числа со знаком отрицательные, если они начинаются с цифр 8, 9, A, B, C, D, E или F, поскольку их двоичные эквиваленты начинаются с 1. Например, число 99h может соответствовать либо десятичному числу 153 (если вы знаете, что имеете дело с однобайтными числами без знака), либо десятичному числу –103 (если вы знаете, что это число со знаком).

Кроме того, байт 99h может соответствовать и десятичному числу 99. Это интересно, но, похоже, противоречит всему, о чем мы говорили до сих пор. В главе 23 объясню, как это работает, а теперь остановимся на памяти.

Глава 16
Сборка памяти

Каждое утро, когда мы просыпаемся, включается наша память. Мы вспоминаем, где находимся, что делали накануне, что планируем сделать сегодня. Память возвращается сразу или фрагментами, и в течение еще нескольких минут человек может чего-то не помнить («Забавно, я не помню, что лег спать в носках»), однако в целом мы способны восстановить непрерывность своей жизни, чтобы начать новый день.

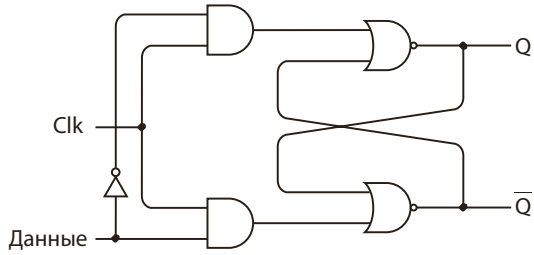
Разумеется, человеческая память не вполне упорядочена. Попытайтесь вспомнить что-нибудь из школьного курса геометрии. Вероятно, начнете думать о сидевшем перед вами однокласснике или о том дне, когда сработала пожарная тревога как раз в тот момент, когда учитель собирался объяснить вам, что значит выражение «что и требовалось доказать».

Человеческая память не является вполне надежной. Письменность, возможно, была изобретена специально, чтобы компенсировать недостатки человеческой памяти. Может быть, прошлой ночью вы внезапно проснулись в 3:00 с отличной идеей для сценария. Схватили ручку и бумагу, которые держите у кровати специально для таких случаев, и записали идею. На следующее утро вы читаете свою блестящую мысль и начинаете работу над сценарием («Парень встречает девушку, погоня на машинах и взрывы»… и это всё?). Или не начинаете.

Мы *пишем*, а затем *читаем*. Мы *сохраняем*, а потом *извлекаем*. Мы *храним* информацию, а в дальнейшем *получаем* к ней *доступ*. Функция памяти заключается в том, чтобы сохранять информацию без искажений между этими двумя событиями. Каждый раз, когда мы сохраняем информацию, мы используем разные типы памяти. Бумага — хороший носитель для хранения текстовой информации, магнитная лента подходит для хранения музыки и фильмов.

Телеграфные реле, будучи объединенными в вентили, а затем в триггеры, также могут хранить информацию. Как мы видели ранее, триггер способен хранить один бит. Это не очень много, но это начало. Как только мы научимся сохранять один бит, мы легко сможем справиться с двумя, тремя и более.

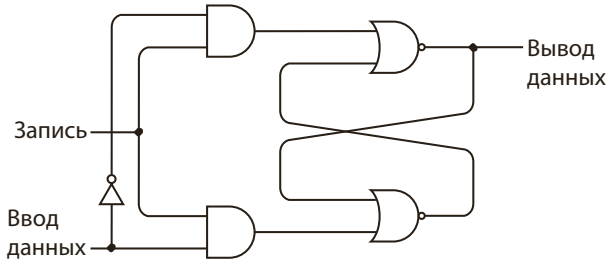
В главе 14 мы познакомились с D-триггером со срабатыванием по уровню, который состоит из инвертора, двух вентилей И и двух вентилей ИЛИ-НЕ.



Когда вход Clk равен 1, выходной сигнал Q совпадает с входным сигналом «Данные». Когда значение входа Clk меняется на 0, выход Q сохраняет последнее значение входа «Данные». Дальнейшие изменения входного сигнала «Данные» не влияют на выходы до тех пор, пока значение входа Clk снова не изменится на 1. Вот таблица логики для триггера.

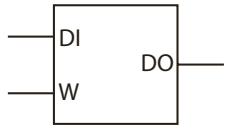
Входы		**Выходы**	
D	**Clk**	**Q**	**$\overline{Q}$**
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

В главе 14 этот триггер использовался в разных схемах, а сейчас он будет применяться исключительно для хранения одного бита. По этой причине я собираюсь переименовать входы и выходы, чтобы они соответствовали цели.

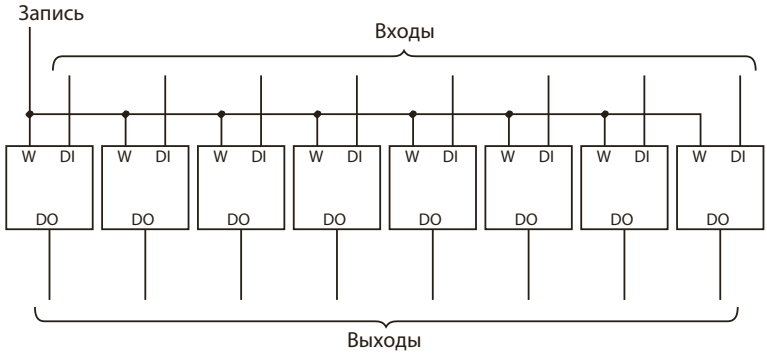


Это тот же триггер, только теперь выход Q называется «Вывод данных» (Data Out, DO), а вход Clk («Запомнить этот бит») стал «Записью» (Write, W). Так же, как мы можем записать некоторую информацию на бумаге, сигнал «Запись» приводит к *записи* или *сохранению* сигнала «Ввод данных» (Data In,

DI) в схеме. Обычно вход «Запись» равен 0, а сигнал «Ввод данных» не влияет на выход. Однако всякий раз, когда мы хотим сохранить значение сигнала «Ввод данных» в триггере, подаем на вход «Запись» 1, а затем снова 0. Как я упоминал в главе 14, схема такого типа также называется *защелкой*, поскольку она как бы «запирает» данные. Вот как мы можем представить однобитную защелку без изображения всех отдельных компонентов.

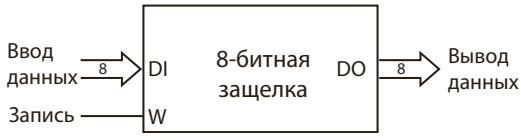


Мы можем достаточно легко объединить несколько однобитных защелок в многобитную. Все, что для этого нужно сделать, — соединить входы «Запись».



Эта 8-битная защелка содержит восемь входов и восемь выходов. Кроме того, защелка имеет один вход под названием «Запись», который обычно равен 0. Чтобы сохранить 8-битное значение в этой защелке, подайте на вход «Запись» 1, а затем 0. Эту защелку также можно изобразить следующим образом.

Или так, чтобы она больше напоминала изображение однобитной.



Другой способ соединения восьми однобитных защелок более сложен. Допустим, нам нужен только один сигнал «Ввод данных» и один сигнал «Вывод данных». Однако мы хотим сохранить значение сигнала «Ввод данных» восемь раз в течение дня или восемь раз в течение следующей минуты. Кроме того, мы хотим иметь возможность в дальнейшем просмотреть эти восемь значений, только бросив взгляд на один сигнал «Вывод данных».

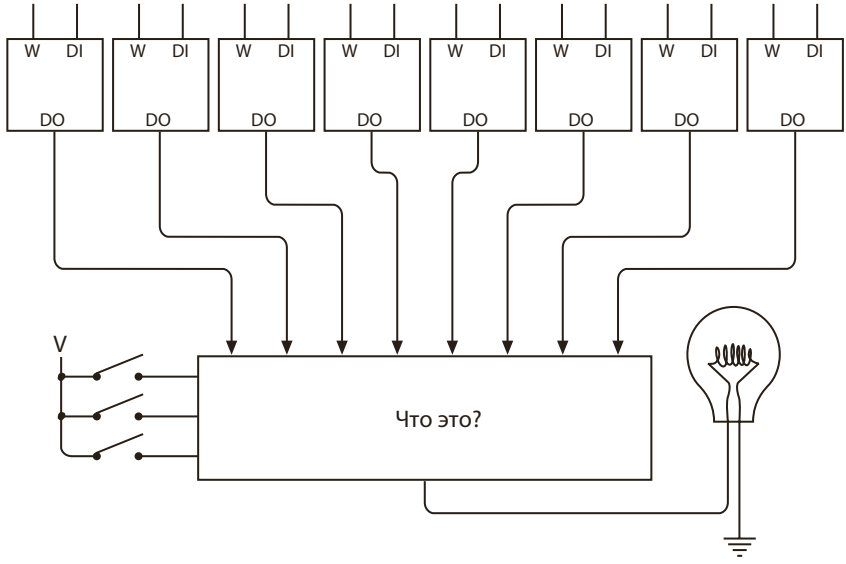
Другими словами, вместо сохранения одного 8-битного значения, как в случае с 8-битной защелкой, мы хотим сохранить восемь отдельных однобитных значений.

Почему мы хотим сделать именно так? Вероятно, потому, что у нас есть только одна лампочка.

Мы знаем, что нам требуется восемь однобитных защелок. Давайте пока не будем волноваться, как именно данные в них сохраняются. Сосредоточимся на проверке сигналов «Вывод данных» этих восьми защелок, используя только одну лампочку. Конечно, мы могли бы проверять выход каждой защелки, *вручную* перенося лампочку от одной защелки к другой, но мы бы предпочли более автоматизированный способ. Фактически мы хотим выбирать одну из восьми однобитных защелок, используя переключатели.

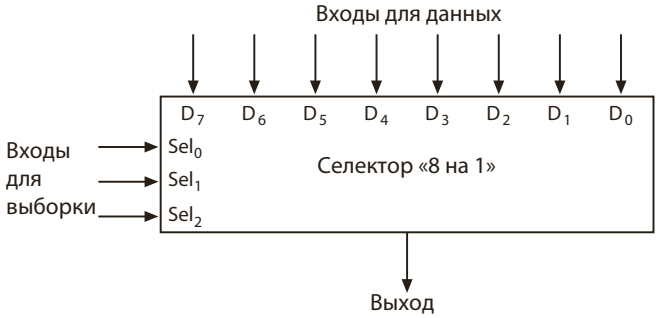
Сколько переключателей нужно? Если хотим выбрать один из восьми элементов, потребуются три переключателя. Три переключателя могут представлять восемь разных значений: 000, 001, 010, 011, 100, 101, 110 и 111.

Итак, вот наши восемь однобитных защелок, три переключателя, лампочка и устройство, которое необходимо поместить между переключателями и лампочкой.



Устройство — это некий корпус с восемью входами сверху и тремя входами слева. Замыкая и размыкая три переключателя, мы можем выбрать, какой из восьми входов должен быть перенаправлен на выход в нижней части корпуса. Этот выход зажигает лампочку.

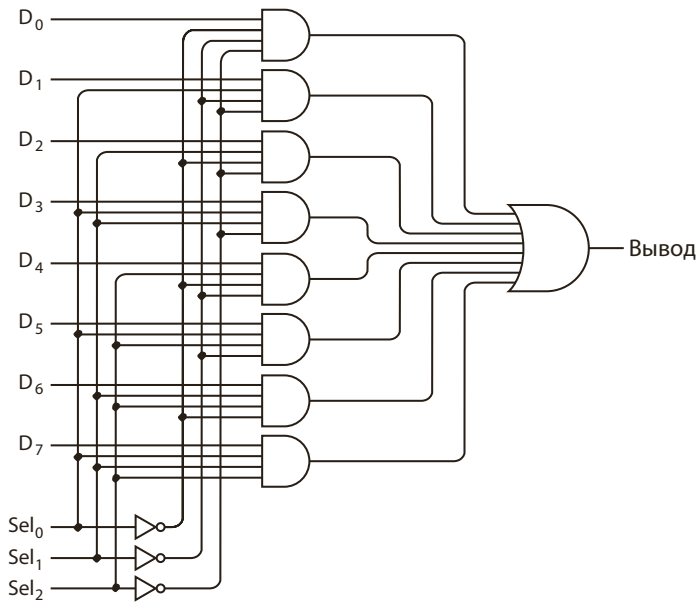
Так что же это за устройство? Мы уже видели что-то подобное, хотя и не с таким количеством входов. Оно похоже на схему, которую мы использовали в главе 14 в первой модифицированной версии сумматора. В том случае нам нужно было нечто позволяющее выбрать, откуда должен поступать сигнал на вход сумматора: от ряда переключателей или с выхода защелки. Тогда мы назвали устройство селектором «2 на 1». В данном случае нам нужен *селектор «8 на 1»*.



Селектор восьми линий на одну имеет восемь входов для данных (вверху) и три входа для выборки (Select, Sel) (слева). Входы Select позволяют выбрать, сигнал какого входа для данных появится на выходе. Например, если сигналы Select равны 000, то выходной сигнал совпадает с D_0. Если входы Select равны 111, то выходной сигнал совпадает с D_7. Если входы Select — 101, выходной сигнал совпадает с D_5. Приведем таблицу логики для этого селектора.

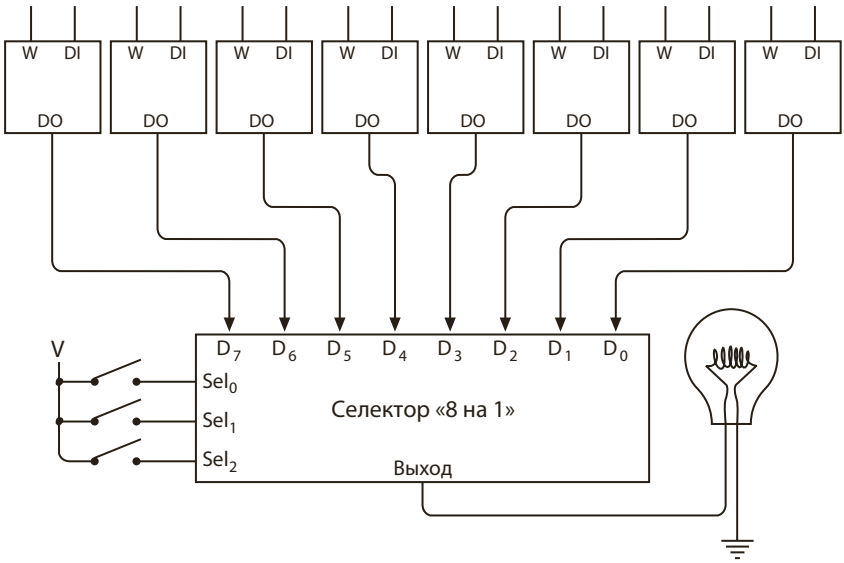
Входы			**Выходы**
Sel_2	**Sel_1**	**Sel_0**	**Q**
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

Селектор «8 на 1» состоит из трех инверторов, восьми четырехвходовых вентилей И и одного восьмивходового вентиля ИЛИ.



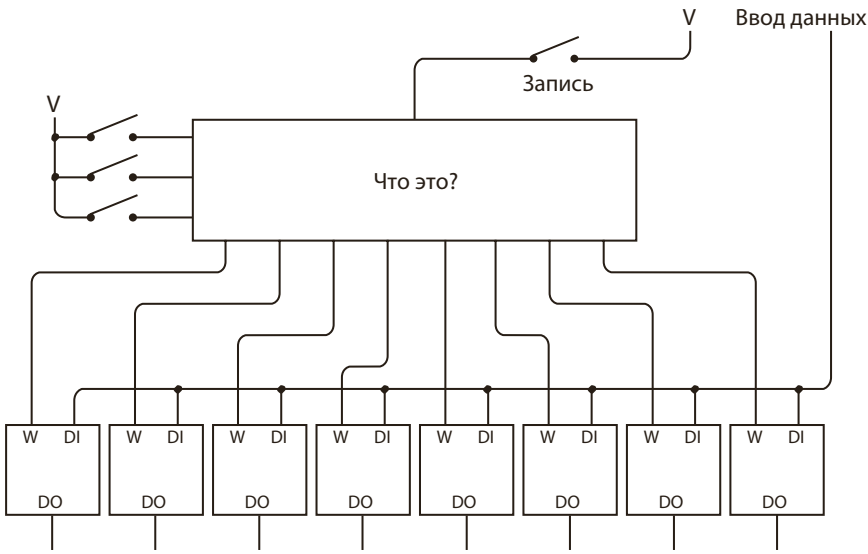
Эта схема может показаться довольно запутанной, однако с помощью следующего примера постараюсь убедить вас, что она работает. Допустим, сигналы Sel_2 и Sel_0 равны 1, а сигнал S_1— 0. На входы шестого сверху вентиля И подаются сигналы Sel_0, Sel_1, Sel_2, каждый из которых равен 1. Ни на один другой вентиль И эти три сигнала не подаются, поэтому выход всех остальных вентилей И будет равен 1. Выход шестого вентиля И — 0 при D_5, равном 0, или 1 при D_5, равном 1. То же касается крайнего справа вентиля ИЛИ. Таким образом, если сигналы для выборки равны 101, выходной сигнал совпадает с сигналом D_5.

Давайте еще раз повторим, чего хотим добиться. Мы пытаемся соединить восемь однобитных защелок так, чтобы в них можно было записывать и считывать данные по отдельности, используя один сигнал «Ввод данных» и один сигнал «Вывод данных». Мы уже выяснили, что можно выбрать сигнал «Вывод данных» одной из восьми защелок, используя селектор «8 на 1».



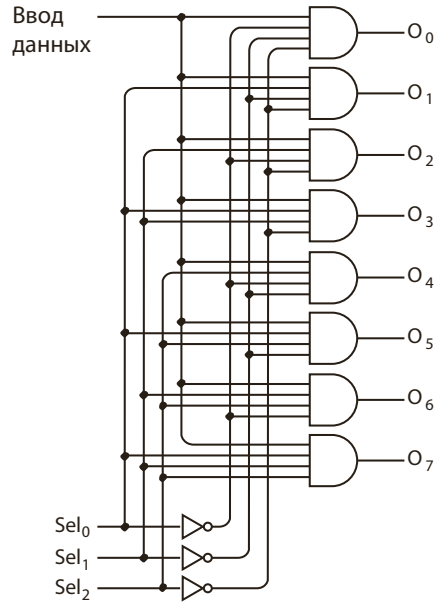
Итак, наша задача наполовину решена. Теперь, когда мы определились с устройством на выходе, давайте займемся входом.

На входе мы имеем сигналы «Данные» и «Запись». Входы «Данные» защелок можно соединить между собой. Однако мы не можем сделать то же самое с сигналами «Запись», поскольку хотим записывать данные отдельно, следовательно, должны подавать один сигнал «Запись» на одну (и только одну!) защелку.



Теперь нужна другая схема, которая немного похожа на селектор «8 на 1», но выполняет прямо противоположное действие. Эта схема называется *дешифратор «3 на 8»*. В главе 11 мы уже видели простой дешифратор данных, когда соединяли переключатели для выбора цвета нашей идеальной кошки.

Дешифратор «3 на 8» имеет восемь выходов. В любой момент все эти выходы равны 0, кроме одного, который выбран входными сигналами Sel_0, Sel_1 и Sel_2. Значение этого выхода совпадает со значением входа «Ввод данных».

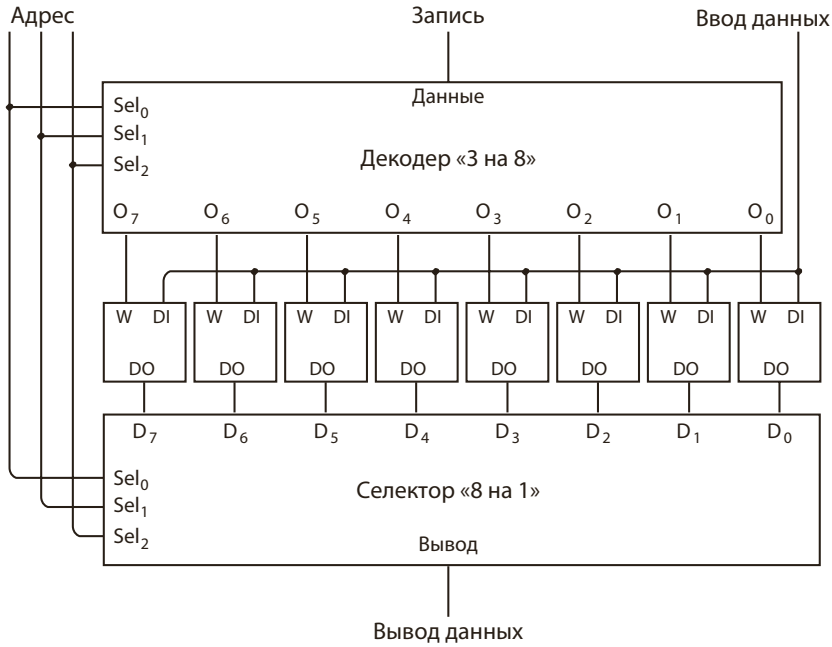


Обратите внимание: входными сигналами шестого вентиля И сверху являются Sel_0, Sel_1, Sel_2. Они не подаются ни на один другой вентиль И, поэтому если на входы для выборки подается значение 101, то выходы всех остальных вентилей И будут равны 0. Вход шестого вентиля И может иметь значение 0, если вход «Ввод данных» равен 0, или 1, если вход «Ввод данных» равен 1. Полная таблица логики имеет следующий вид.

Входы			**Выходы**							
Sel_1	**Sel_2**	**Sel_3**	**O_7**	**O_6**	**O_5**	**O_4**	**O_3**	**O_2**	**O_1**	**O_0**
0	0	0	0	0	0	0	0	0	0	Данные
0	0	1	0	0	0	0	0	0	Данные	0
0	1	0	0	0	0	0	0	Данные	0	0
0	1	1	0	0	0	0	Данные	0	0	0
1	0	0	0	0	0	Данные	0	0	0	0

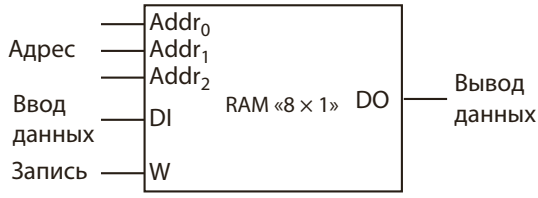
Входы			Выходы							
Sel_1	Sel_2	Sel_3	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
1	0	1	0	0	Данные	0	0	0	0	0
1	1	0	0	Данные	0	0	0	0	0	0
1	1	1	Данные	0	0	0	0	0	0	0

Полная схема с восемью защелками выглядит таким образом.



Важно: три сигнала выборки для дешифратора и селектора являются одинаковыми (они обозначены словом «адрес» — address, Addr). Подобно почтовому индексу, этот 3-битный адрес определяет, к какой из восьми однобитных защелок мы обращаемся. На входе сигнал «Адрес» определяет, какая защелка сохранит сигнал «Данные» под воздействием сигнала «Запись». На выходе (в нижней части схемы) вход «Адрес» управляет селектором «8 на 1» для того, чтобы считать выходной сигнал одной из восьми защелок.

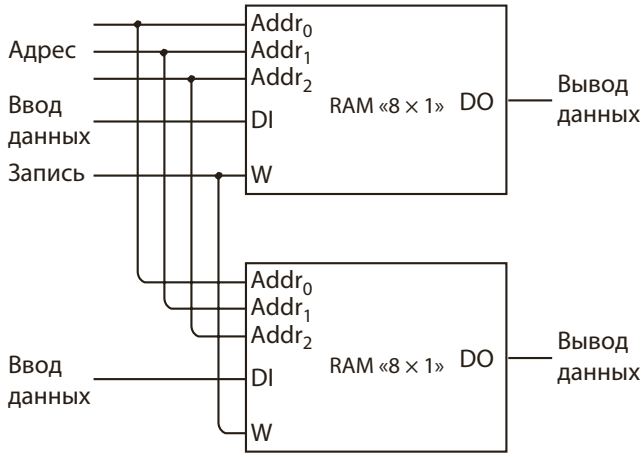
Эта конфигурация защелок иногда называется *памятью с записью/чтением*, но чаще — *памятью с произвольным доступом*, или *произвольной выборкой* (random access memory, RAM). Эта конкретная конфигурация RAM хранит восемь отдельных однобитных значений. Ее можно изобразить следующим образом.



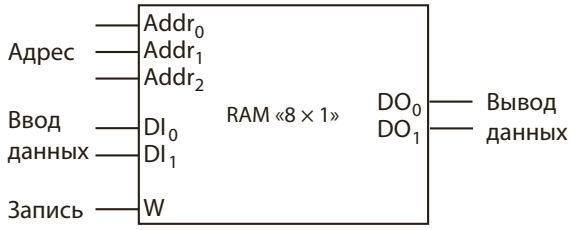
Устройство называется *памятью*, потому что оно сохраняет информацию. Возможность *чтения/записи* говорит о том, что вы можете сохранить новое значение в любой защелке (*записать* значение), а также узнать, что хранится в каждой из защелок (*прочитать* значение). Термин «произвольный доступ» означает, что запись и считывание информации из защелок могут осуществляться путем изменения входных сигналов «Адрес». Кроме памяти с произвольным доступом, существует память с последовательным доступом, при использовании которой для считывания значения, хранящегося по адресу 101, требуется сначала прочитать значение, хранящееся по адресу 100.

Описанная выше конфигурация RAM часто называется *массивом RAM*. Этот конкретный массив RAM организован по схеме, иногда сокращенно обозначаемой «8 × 1» — восемь однобитных значений. Чтобы определить общее количество битов, которые можно сохранить в массиве RAM, нужно перемножить эти два числа.

Массивы RAM можно комбинировать. Например, объединить два массива RAM «8 × 1».

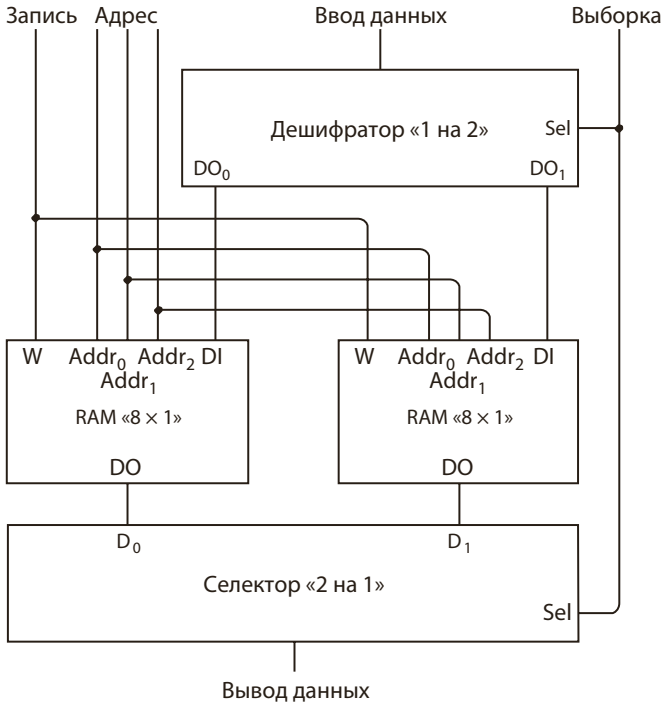


В данном случае входы «Адрес» и «Запись» двух массивов RAM «8 × 1» соединены, поэтому в результате получается массив RAM «8 × 2».

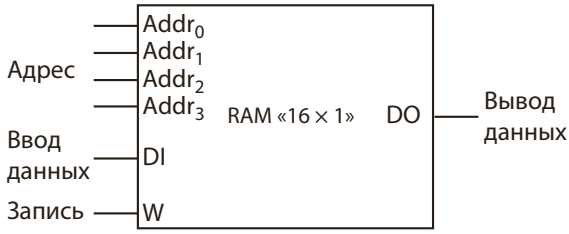


Этот массив RAM хранит восемь значений, размер каждого из которых составляет два бита.

Кроме того, два массива RAM «8 × 1» можно объединить как отдельные защелки, используя селектор «2 на 1» и дешифратор «1 на 2».



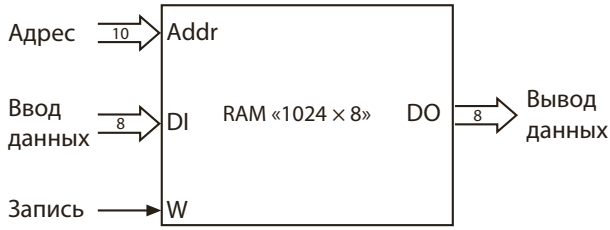
Сигнал «Выборка», который подается как на дешифратор, так и на селектор, по сути, выбирает один из двух массивов RAM «8 × 1». На самом деле он является четвертой адресной линией. Таким образом, мы имеем дело с массивом RAM «16 × 1».



Этот массив RAM хранит 16 значений, размер каждого из которых составляет один бит. Количество значений, хранящихся в массиве RAM, напрямую зависит от количества входов «Адрес». В отсутствие таких входов (как в случае с однобитной и 8-битной защелками) может быть сохранено только одно значение. При наличии одного входа «Адрес» можно сохранить два значения. Два входа «Адрес» позволяют хранить четыре значения, три входа «Адрес» — восемь, четыре входа — шестнадцать. Такое отношение можно выразить с помощью уравнения:

$$\text{Количество значений в массиве RAM} = 2^{\text{количество входов «Адрес»}}.$$

Я показал, как можно сконструировать небольшие массивы RAM, поэтому вам нетрудно будет представить гораздо более крупные. Например, такой.



Этот массив RAM хранит в общей сложности 8196 бит информации, которые организованы в виде 1024 значений по восемь бит каждое. Этот массив имеет десять входов «Адрес», так как 2^{10} равно 1024, восемь входов и восемь выходов для данных.

Другими словами, этот массив RAM хранит 1024 байт. Он похож на почтовое отделение с 1024 абонентскими ящиками. В каждом из них хранится значение размером один байт (которое, правда, может представлять просто спам).

Одна тысяча двадцать четыре байта — *килобайт*, и здесь возникает большая путаница. Чаще всего приставка «*кило-*» (от греческого «тысяча») используется в метрической системе. Например, килограмм — это 1000 граммов, километр — 1000 метров. Однако килобайт составляет 1024 байт, а не 1000 байт.

Проблема в том, что метрическая система основана на степенях 10, а двоичные числа — на степенях 2, и этим системам никогда не сойтись. Степенями 10 являются 10, 100, 1000, 10 000, 100 000 и т. д., а степенями 2 — 2, 4, 8, 16, 32, 64 и т. д. Не существует степени 10, которая была бы равна некоторой степени 2.

Однако время от времени эти две системы сближаются. Да, значение 1000 достаточно близко к значению 1024. Выражаясь математическим языком, 2 в степени 10 «приблизительно равно» 10 в степени 3:

$$2^{10} \approx 10^3.$$

В этом соотношении нет ничего волшебного. Оно всего лишь подразумевает, что конкретная степень 2 приблизительно равна конкретной степени 10. Это совпадение позволяет людям использовать термин «килобайт памяти», подразумевая 1024 байта.

Килобайт сокращенно обозначается Кбайт (международное обозначение — Kb). Описанный выше массив RAM может хранить 1024 байт, или один килобайт (1 Кбайт).

Мы не подразумеваем, что в массиве RAM емкостью один килобайт хранится 1000 байт. В нем хранится больше тысячи байт, а именно 1024. Чтобы произвести впечатление знающего человека, следует говорить «один килобайт».

Один килобайт памяти имеет восемь входов и восемь выходов для данных, а также десять входов «Адрес». Поскольку доступ к байтам осуществляется с помощью десяти входов «Адрес», массив RAM хранит 2^{10} байт. Всякий раз, когда добавляем еще один вход «Адрес», мы удваиваем объем памяти. Каждая строка следующей последовательности представляет собой удвоение памяти:

1 килобайт = 1024 байт = 2^{10} байт ≈ 10^3 байт;
2 килобайта = 2048 байт = 2^{11} байт;
4 килобайта = 4096 байт = 2^{12} байт;
8 килобайт = 8192 байт = 2^{13} байт;
16 килобайт = 16 384 байт = 2^{14} байт;
32 килобайта = 32 768 байт = 2^{15} байт;
64 килобайта = 65 536 байт = 2^{16} байт;
128 килобайт = 131 072 байт = 2^{17} байт;
256 килобайт = 262 144 байт = 2^{18} байт;
512 килобайт = 524 288 байт = 2^{19} байт;
1024 килобайт = 1 048 576 байт = 2^{20} байт ≈ 10^6 байт.

Обратите внимание: указанные слева количества килобайтов — степени 2.

Ту же логику, которая позволяет называть 1024 байт килобайтом, мы можем использовать для того, чтобы назвать 1024 килобайт *мегабайтом* (приставка «мега-» — от греческого «великий»). Мегабайт сокращенно обозначается Мбайт (Mbyte, реже MB). Удвоение памяти продолжается:

1 мегабайт = 1 048 576 байт = 2^{20} байт ≈ 10^6 байт;
2 мегабайта = 2 097 152 байт = 2^{21} байт;
4 мегабайта = 4 194 304 байт = 2^{22} байт;
8 мегабайт = 8 388 608 байт = 2^{23} байт;
16 мегабайт = 16 777 216 байт = 2^{24} байт;
32 мегабайта = 33 554 432 байт = 2^{25} байт;
64 мегабайта = 67 108 864 байт = 2^{26} байт;
128 мегабайт = 134 217 728 байт = 2^{27} байт;
256 мегабайт = 268 435 456 байт = 2^{28} байт;
512 мегабайт = 536 870 912 байт = 2^{29} байт;
1024 мегабайт = 1 073 741 824 байт = 2^{30} байт ≈ 10^9 байт.

Одна тысяча двадцать четыре мегабайта составляют *гигабайт* (приставка «гига-» — от греческого «гигантский»), который обозначается буквами Гб (GB).

Один *терабайт* (от греческого «чудовищный») равен 2^{40} байт (приблизительно 10^{12}), или 1 099 511 627 776 байт. Терабайт обозначается буквами Тб (TB).

Килобайт равен примерно тысяче байтов, мегабайт — миллиону, гигабайт — миллиарду, терабайт — триллиону байтов.

Петабайт равен 2^{50}, или 1 125 899 906 842 624 байт (приблизительно 10^{15}, или квадриллион), а *экзабайт* равен 2^{60}, или 1 152 921 504 606 846 976 байт (приблизительно 10^{18}, или квинтиллион).

Примите к сведению, что домашние компьютеры, купленные в период написания этой книги (1999 год), обычно имели 32 или 64 (иногда 128) мегабайта памяти с произвольным доступом*. (Заметьте, я говорю о памяти RAM, а не о емкости жестких дисков.) Это 33 554 432 байт, или 67 108 864 байт, или 134 217 728 байт.

Разумеется, в разговоре люди используют сокращения. Обладатель 65 536 байтов памяти скажет: «У меня 64 килобайта (и я гость из далекого 1980 года)». Пользователь компьютера с памятью 33 554 432 байт уточнит: «У меня 32 мега». А везунчик, имеющий 1 073 741 824 байт памяти, подчеркнет: «У меня целый гиг».

---

* Современные игровые ПК могут иметь вплоть до 64 гигабайт (и даже более) оперативной памяти. *Прим. науч. ред.*

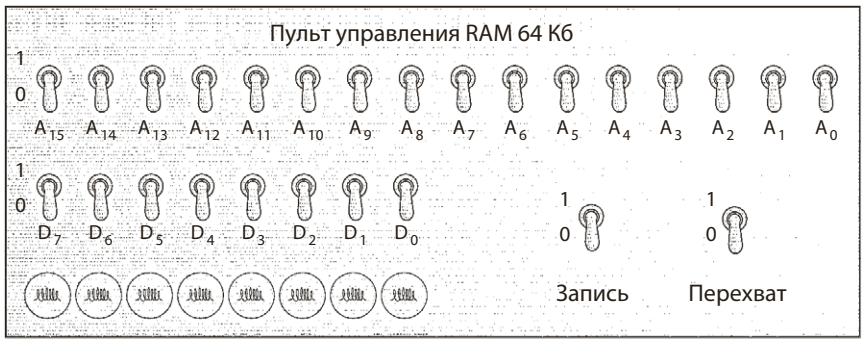
Иногда упоминают *килобиты* или *мегабиты* (обратите внимание на использование слова «биты» вместо «байты»), но это бывает редко. Почти всегда, когда речь идет о памяти, сообщают количество байтов, а не битов. (Чтобы преобразовать байты в биты, нужно умножить их на 8.) Обычно когда в разговоре упоминаются килобиты или мегабиты, они имеют отношение к скорости передачи данных по кабелю и используются в таких словосочетаниях, как «килобиты в секунду» и «мегабиты в секунду». Например, когда речь идет о модеме 56 K, имеется в виду именно 56 килобит в секунду, а не килобайт; технология Gigabit Ethernet обеспечивает передачу одного гигабита информации в секунду и т. д.

Теперь, научившись собирать массивы RAM любого нужного размера, постараемся не сильно увлекаться. Давайте просто представим массив RAM емкостью 65 536 байт.

Почему 64 килобайт? Почему не 32 килобайт или не 128 килобайт? *Потому что 65 536 — хорошее круглое число*. Оно равно 2^{16}. Этот массив RAM имеет 16-битный адрес. Другими словами, этот адрес равен двум байтам. В шестнадцатеричной системе счисления значение этого адреса находится в диапазоне от 0000h до FFFFh.

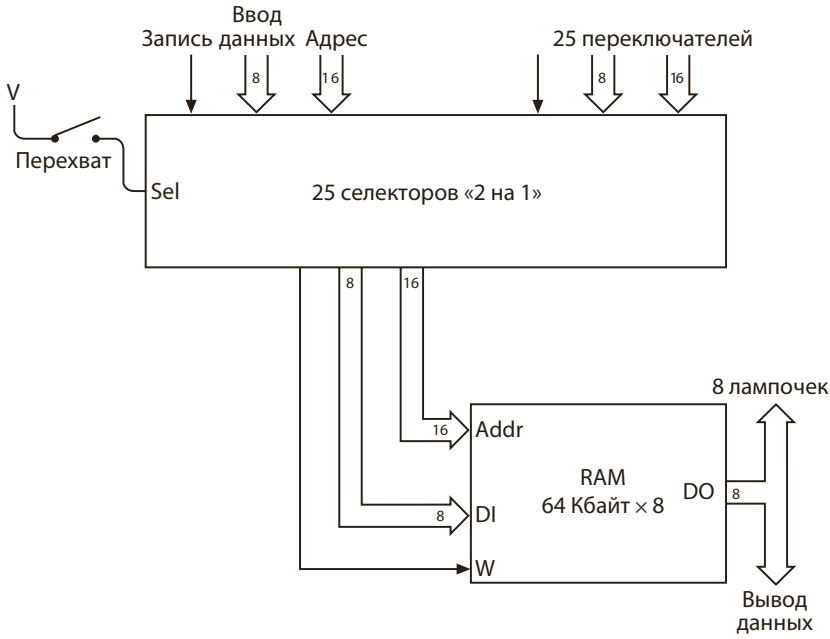
Ранее я намекнул на то, что объем памяти, равный 64 килобайт, был характерен для персональных компьютеров, купленных в 1980-х годах, хотя эта память собиралась и не из телеграфных реле. Удастся ли сконструировать такое устройство, используя реле? Надеюсь, вы не рассматриваете такую возможность всерьез. Наша конструкция предполагает использование девяти реле для каждого бита памяти, поэтому для создания массива RAM 64 Кб × 8 потребуются почти пять миллионов реле.

Нам также не помешает пульт управления, позволяющий пользоваться этой памятью: записывать значения или считывать их. Такой пульт должен иметь 16 переключателей для указания адреса, восемь переключателей для задания 8-битного значения, которое мы хотим записать, еще одного переключателя для самого сигнала записи и восемь лампочек для отображения определенного 8-битного значения.



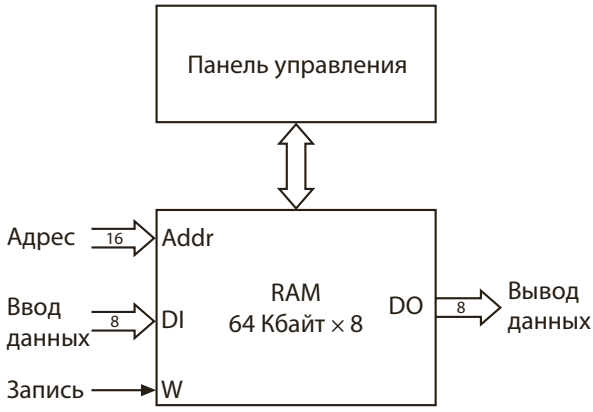
Все переключатели изображены в положении «Выключено» (0). Я также добавил переключатель с надписью «Перехват». Цель этого переключателя — позволить другим схемам использовать ту же память, к которой подключен пульт управления. Когда этот переключатель установлен в положение 0 (как показано на рисунке), остальные переключатели пульта управления ничего не делают. Однако когда переключатель установлен в положение 1, управление памятью осуществляется исключительно с помощью пульта.

Для создания переключателя «Перехват» потребуется набор селекторов «2 на 1», а именно 25 штук: шестнадцать — для сигналов «Адрес», восемь — для переключателей, позволяющих вводить данные, и еще один — для переключателя «Запись». Ниже показана схема.



Когда переключатель «Перехват» разомкнут (как показано на рисунке), на входы массива RAM 64 Кбайт × 8 «Адрес», «Данные» и «Запись» поступают сигналы извне, как показано над левым верхним углом блока селекторов «2 на 1». Когда переключатель «Перехват» замкнут, сигналы на входы массива RAM «Адрес», «Данные» и «Запись» поступают от переключателей на пульте управления. В любом случае сигналы «Вывод данных» из массива RAM поступают на восемь лампочек и, вероятно, куда-нибудь еще.

Массив RAM 64 Кбайт × 8 с таким пультом управления можно изобразить следующим образом.



Когда переключатель «Перехват» замкнут, вы можете использовать 16 переключателей «Адрес» для выбора любого из 65 536 адресов. Лампочки показывают 8-битное значение, которое в настоящий момент хранится в памяти по этому адресу. Вы можете использовать восемь переключателей «Данные» для задания нового значения, а записать это значение в память с помощью переключателя «Запись».

Массив RAM 64 Кб × 8 и пульт управления, безусловно, помогут вам отследить любое из 65 536 8-битных значений, которое может понадобиться. Однако эта конструкция также позволяет некоторым другим схемам использовать значения, которые мы сохранили в памяти, и записывать в нее новые.

Есть еще одна *важная* деталь, которую необходимо знать о памяти: когда в главе 11 вы познакомились с концепцией вентилей, я перестал рисовать отдельные реле, из которых состоят эти вентили. В частности, с тех пор я больше не указывал на схемах, что каждое реле подключено к какому-то источнику питания. Всякий раз при срабатывании реле электричество течет через катушку электромагнита и удерживает металлический контакт на месте.

Итак, у вас есть массив RAM 64 Кб × 8, весь объем которого заполнен вашими любимыми байтами. Что произойдет, если вы отключите его от источника питания? Все электромагниты потеряют свои магнитные свойства, и с громким щелчком все контакты реле вернутся в свое исходное положение. А содержимое этой памяти? Оно исчезнет навсегда!

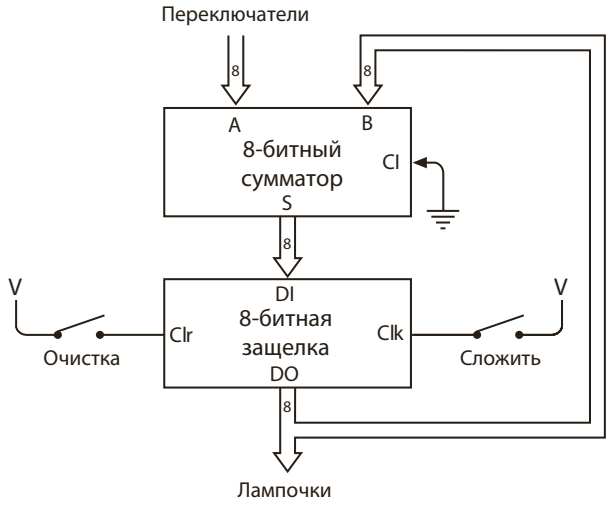
Вот почему память с произвольным доступом также называется *энергозависимой*. Для хранения содержимого ей требуется постоянное энергоснабжение.

Глава 17
Автоматизация

Человек часто демонстрирует удивительную находчивость и усердие, но в то же время ему присуща страшная лень. Всем известно, что люди не любят работать. Наше отвращение к работе так сильно, а изобретательность настолько выражена, что мы готовы потратить бесчисленные часы на проектирование и сборку устройств, которые могли бы урезать наш рабочий день на несколько минут. И мало что может доставить человеку больше удовольствия, чем виде́ние, в котором он расслабленно качается в гамаке, наблюдая, как его новенькое изобретение косит газон. Боюсь, на следующих страницах вы не найдете чертежей автоматической газонокосилки. Однако в этой главе будет описан ряд устройств в порядке возрастания сложности, позволяющих автоматизировать процесс сложения и вычитания чисел. Понимаю, что подобное заявление вряд ли поразит вас. К концу этой главы мы соберем машину, которая сможет решить практически любую задачу, предполагающую выполнение сложения и вычитания, а это действительно весьма обширный класс.

Разумеется, чем устройство заковыристее, тем оно сложнее, поэтому часть информации может показаться трудной. Никто не будет винить вас, если пропустите излишне сложные фрагменты. Время от времени у вас будет возникать желание взбунтоваться и навсегда отказаться от попыток решить математическую задачу с помощью электрических и механических устройств. Обязательно следите за моей мыслью, потому что к концу главы мы изобретем машину, которую с полным правом можно назвать *компьютером*.

Последний рассмотренный нами сумматор был описан в главе 14. Эта версия включала 8-битную защелку, где накапливалась общая сумма чисел, введенных с помощью набора из восьми переключателей.



Как вы помните, 8-битная защелка использует триггеры для хранения 8-битного значения. Чтобы использовать это устройство, на мгновение нажмите кнопку «Очистка» для обнуления содержимого защелки. Затем с помощью переключателей введите первое число. Сумматор просто прибавит его к нулевому значению на выходе защелки, поэтому результатом будет введенное число. При нажатии кнопки «Сложить» это число сохранится в защелке и отобразится лампочками. Теперь с помощью переключателей введите второе число. Сумматор прибавит его к уже сохраненному в защелке.

Нажатие кнопки «Сложить» снова приводит к сохранению общей суммы в защелке и ее отображению с помощью лампочек. Так вы можете сложить множество чисел, отображая общую сумму, которая, правда, ограничена числом 255 при использовании восьми лампочек.

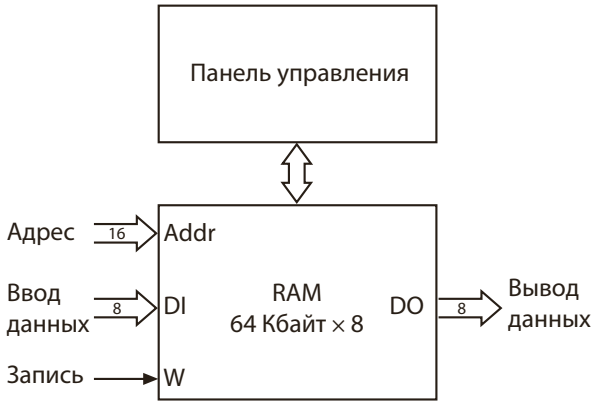
Когда я продемонстрировал эту схему в главе 14, вам было известно только о *защелках со срабатыванием по уровню*. Для сохранения данных в такой защелке нужно, чтобы значение входного сигнала Clk стало равным 1, а затем 0. Пока вход Clk — 1, сигналы на входах защелки для данных могут меняться, и эти изменения будут влиять на ее содержимое. Чуть позже я познакомил вас с защелками *со срабатыванием по фронту*, которые сохраняют данные в краткий промежуток времени, пока входной сигнал Clk изменяется с 0 на 1. Защелки со срабатыванием по фронту немного проще, поэтому в этой главе при описании защелок буду предполагать, что речь идет о защелках именно этого типа.

Защелка, применяемая для хранения текущего значения суммы, называется *аккумулятором* или *накопителем*. Однако далее мы увидим, что аккумулятору нет необходимости просто накапливать сумму. Часто это защелка, где

сначала сохраняется одно число, а затем результат сложения или вычитания из него другого числа.

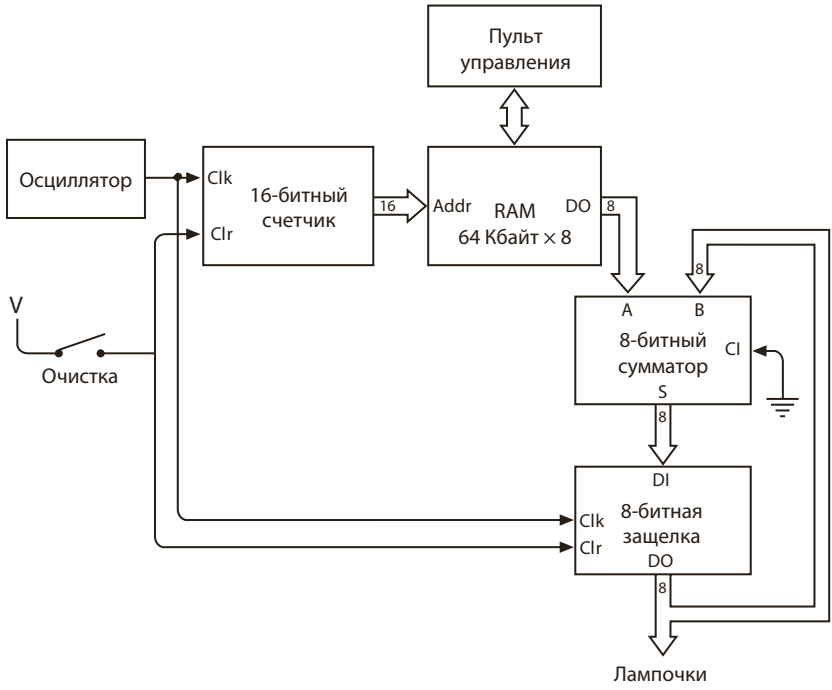
У показанной выше счетной машины существует серьезный недостаток, который сразу бросается в глаза. Допустим, существует список из 100 двоичных чисел, которые вы хотите сложить. Вы садитесь перед сумматором, старательно вводите каждое число и накапливаете сумму. По окончании процедуры обнаруживаете, что несколько чисел в списке были неправильными. В этом случае вам придется начинать все сначала.

А может, и нет. В предыдущей главе мы использовали около пяти миллионов реле для создания массива RAM емкостью 64 килобайта. Кроме того, мы собрали пульт управления, который позволяет замкнуть переключатель «Перехват» и фактически перехватить управление массивом RAM, чтобы производить запись и чтение данных с помощью переключателей.



Если бы вы ввели все 100 двоичных чисел в этот массив RAM, а не напрямую в сумматор, то сделать несколько исправлений было бы проще.

Итак, теперь перед нами стоит задача подключить массив RAM к сумматору, аккумулирующему итоговое значение. Очевидно, что вместо сигналов от переключателей на вход сумматора можно подать выходные сигналы массива RAM (DO), однако легко упустить тот факт, что 16-битный счетчик (вроде собранного в главе 14) способен управлять адресными сигналами. Сигналы массива RAM «Ввод данных» (DI) и «Запись» в этой схеме отсутствуют за ненадобностью.



Разумеется, это не самое простое счетное устройство. Перед его использованием нужно замкнуть переключатель «Очистка». При этом содержимое защелки обнуляется, а для выходного сигнала 16-битного счетчика задается значение 0000h. Затем вы замыкаете переключатель «Перехват» на пульте управления массивом RAM, после чего можете записать в память набор подлежащих сложению 8-битных чисел, начиная с адреса 0000h. Если собираетесь суммировать 100 чисел, сохраните их в ячейках памяти с адресами 0000h–0063h. Кроме того, вы должны записать значение 00h во все неиспользуемые ячейки памяти. Затем можете разомкнуть переключатель «Перехват» на пульте управления массивом RAM, чтобы снова передать сумматору управление памятью, а также разомкнуть переключатель «Очистка». Теперь можно расслабиться и любоваться мигающими лампочками.

Вот что происходит при совершении вышеописанных действий: при первом размыкании переключателя «Очистка» текущим адресом массива RAM является 0000h. Восьмибитное значение, сохраненное в ячейке по этому адресу, подается на один из входов сумматора. На другой вход подается значение 00h, поскольку содержимое защелки также обнулено.

Осциллятор генерирует тактовый сигнал, или синхросигнал, который быстро колеблется между значениями 0 и 1. После размыкания переключателя

«Очистка» при каждом изменении синхросигнала с 0 на 1 одновременно происходят две вещи: в защелке сохраняется сумма из сумматора, а значение 16-битного счетчика увеличивается на единицу, то есть происходит обращение к следующему адресу в массиве RAM. При первом изменении значения синхросигнала с 0 на 1 после размыкания переключателя «Очистка» в защелке сохраняется первое число, а значение счетчика изменяется на 0001h. Во второй раз в защелке сохраняется сумма первого и второго чисел, а значение счетчика изменяется на 0002h. И так далее.

Разумеется, здесь я делаю некоторые допущения. Прежде всего предполагается, что частота колебаний осциллятора достаточно низкая, чтобы остальные компоненты схемы успевали сработать. При каждом изменении значения синхросигнала одни многочисленные реле должны активировать другие, чтобы на выходе сумматора появилось значение суммы.

Одна из проблем этой схемы в том, что мы не можем остановить ее работу. В какой-то момент лампочки перестанут мигать потому, что в оставшихся ячейках памяти хранится значение 00h. Тогда вы сможете считать двоичную сумму. Когда счетчик достигнет значения FFFFh, он *обнулится* (подобно одометру автомобиля), и автоматический сумматор снова начнет прибавлять числа к полученной ранее сумме.

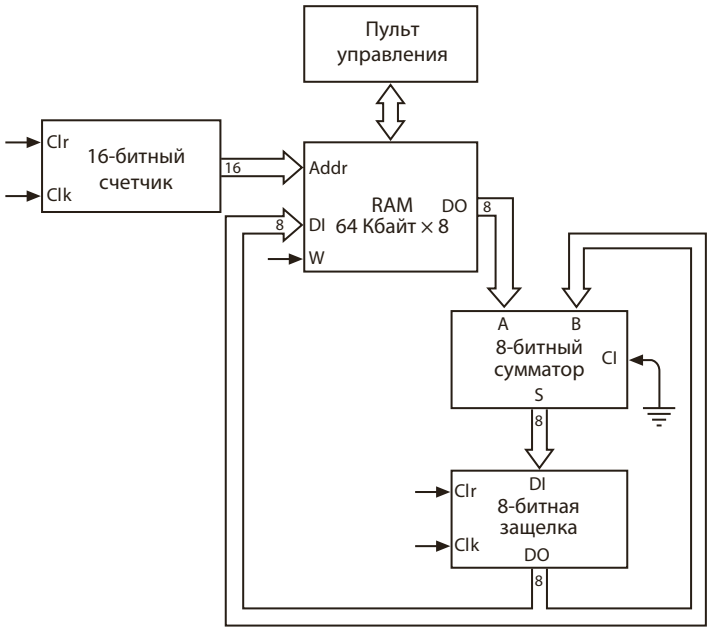
У этого сумматора есть и другие недостатки. Он способен производить только сложение 8-битных чисел. Мало того, что в массиве RAM нельзя сохранить число больше 255, — сама сумма также ограничена значением 255. Кроме того, наш сумматор не способен производить вычитание, хотя вы можете выразить отрицательные числа с помощью дополнения до двух, в случае чего сумматор будет обрабатывать числа только в диапазоне от –128 до 127. Очевидно, что для суммирования больших значений (например, 16-битных) требуется удвоить ширину массива RAM, сумматора и защелки, а также добавить еще восемь лампочек. Однако к таким инвестициям вы можете оказаться не готовы.

Конечно, я бы даже не упомянул об этих проблемах, если бы не был уверен, что они решаемы. Осложнение, на котором я хотел бы сосредоточить ваше внимание в первую очередь, заключается в другом. Что, если вас не интересует одна сумма 100 чисел и вы хотите использовать сумматор для сложения 50 пар чисел и получения 50 разных сумм? Что, если вам нужна машина, способная складывать числа группами по два, десять и т. д., а также предоставлять удобный доступ к результатам своей работы?

Показанный ранее автоматизированный сумматор отображает текущую сумму с помощью набора лампочек, подключенных к защелке. Такая схема не подойдет, если вы решите сложить 50 пар чисел, чтобы получить 50 разных сумм. Вместо этого вы, вероятно, захотите, чтобы результаты сохранялись в массиве

RAM. Тогда вы можете использовать пульт управления памятью для проверки результатов в удобное для вас время. Специально для этой цели такой пульт управления предусматривает дополнительный набор лампочек.

Получается, мы можем избавиться от лампочек, подключенных к защелке. Вместо этого выход защелки необходимо подключить ко входу DI массива RAM для записи сумм в память.



На этой схеме отсутствуют некоторые другие компоненты сумматора, в частности осциллятор и переключатель «Очистка». Я убрал их, поскольку уже не столь очевидно, откуда на входы счетчика и защелки поступают сигналы Clr и Clk. Более того, теперь, когда мы задействовали входы DI массива RAM, нам нужен способ управления его сигналом «Запись».

Давайте на мгновение оставим схему и сосредоточимся на стоящей перед нами задаче. Итак, мы хотим сконструировать сумматор, возможности которого не ограничиваются сохранением текущей суммы складываемых чисел, хотим полностью контролировать количество слагаемых, а также количество разных сумм, сохраняемых в памяти для последующего изучения.

Предположим, нам требуется сложить три числа, потом два, а затем еще три. Мы могли бы сохранить эти числа в массиве RAM, начиная с адреса 0000h, чтобы содержимое памяти выглядело следующим образом.

Таким образом я буду представлять раздел памяти. Прямоугольники — это ячейки памяти. Каждый байт находится в ячейке. Адрес ячейки указан слева. Нет необходимости указывать все адреса, поскольку они идут по порядку, и мы всегда можем выяснить, какой адрес соответствует той или иной ячейке. Приведенные справа комментарии указывают, что сумматор должен сохранить три суммы в пустых ячейках. (Несмотря на то что в этих прямоугольниках ничего нет, ячейки памяти необязательно пустые. В памяти всегда *что-то* содержится, даже если это просто случайные данные. Правда, в настоящий момент в ней нет ничего полезного.)

Знаю, вам не терпится попрактиковаться в сложении шестнадцатеричных чисел и самостоятельно заполнить пустые прямоугольники. Однако цель такой демонстрации не в этом. Нужно, чтобы автоматизированный сумматор сделал всю работу за нас.

Первая версия сумматора выполняла единственное действие — сложение содержимого ячейки памяти со значением в 8-битной защелке, которую я назвал аккумулятором. Теперь необходимо, чтобы сумматор выполнял *четыре различных действия*. Для начала суммирования потребуется, чтобы сумматор передал байт из памяти в аккумулятор.

Назовем эту операцию *загрузкой*. Вторая операция — *сложение* байта из памяти с содержимым аккумулятора, третья — *сохранение* в памяти суммы из аккумулятора. В конце нам нужно каким-то образом *остановить* работу сумматора.

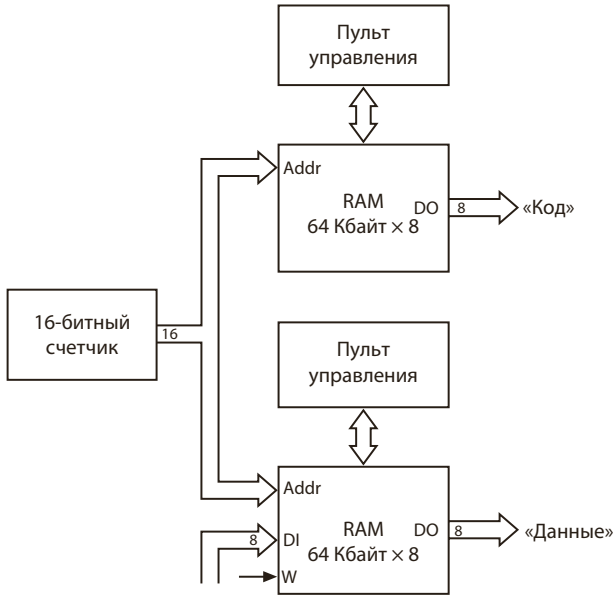
Давайте подробно распишем действия, которые должен выполнить сумматор в данном конкретном примере:

— *загрузить* значение из ячейки 0000h в аккумулятор;
— *сложить* значение из ячейки 0001h со значением в аккумуляторе;
— *сложить* значение из ячейки 0002h со значением в аккумуляторе;
— *сохранить* содержимое аккумулятора в ячейке 0003h;
— *загрузить* значение из ячейки 0004h в аккумулятор;
— *сложить* значение из ячейки 0005h со значением в аккумуляторе;
— *сохранить* содержимое аккумулятора в ячейке 0006h;
— *загрузить* значение из ячейки 0007h в аккумулятор;
— *сложить* значение из ячейки 0008h со значением в аккумуляторе;
— *сложить* значение из ячейки 0009h со значением в аккумуляторе;
— *сохранить* содержимое аккумулятора в ячейке 000Ah;
— *остановить* работу сумматора.

Обратите внимание: как и в исходной версии сумматора, адресация байтов памяти происходит последовательно, начиная с ячейки 0000h. Исходный сумматор просто прибавлял содержимое ячейки памяти по этому адресу к содержимому аккумулятора. В некоторых случаях это действие по-прежнему уместно. Однако иногда нам требуется *загрузить* значение из памяти непосредственно в аккумулятор или *сохранить* содержимое аккумулятора в памяти. После выполнения всех этих операций мы хотим, чтобы сумматор прекратил работу и можно было бы проверить содержимое памяти.

Как это реализовать? Важно понимать, что мы не можем просто записать в память кучу чисел и ожидать, что сумматор догадается, что с ними делать. Для каждого числа в массиве RAM необходимо предусмотреть некий числовой код, соответствующий операциям «*Загрузка*», «*Сложение*», «*Сохранение*» и «*Остановка*».

Вероятно, проще всего (и, разумеется, затратнее) хранить эти коды в отдельном массиве RAM. Доступ к этому второму массиву осуществляется одновременно с доступом к исходному массиву. Однако вместо слагаемых он будет содержать коды, указывающие на то, что должен сделать сумматор с содержимым соответствующей ячейки исходного массива RAM. Исходный массив можно обозначить словом «Данные», а новый массив — словом «Код».



Мы уже выяснили, что наш новый сумматор должен записывать суммы в исходный массив «Данные». Однако в новый массив «Код» значения будут сохраняться только при помощи пульта управления.

Нам нужны четыре кода для обозначения четырех действий, которые должен выполнять новый сумматор. Мы можем назначить этим действиям любые коды, например следующие.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Остановить	FFh

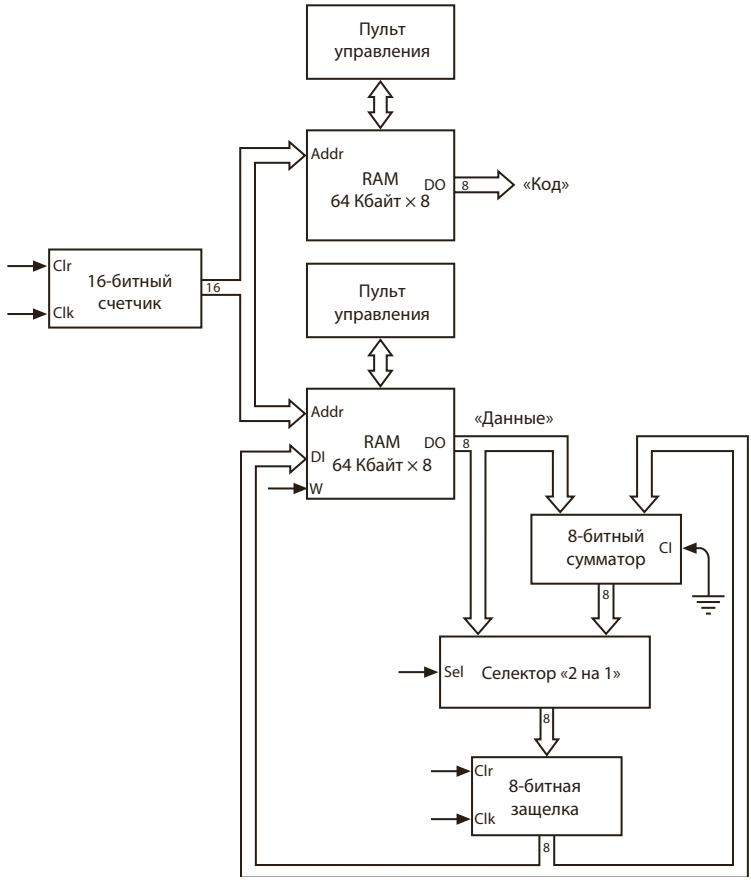
Итак, чтобы выполнить три операции сложения из приведенного выше примера, нужно использовать пульт управления для сохранения следующих значений в массиве «Код».

0000h:	10h	Загрузить
	20h	Сложить
	20h	Сложить
	11h	Сохранить
0004h:	10h	Загрузить
	20h	Сложить
	11h	Сохранить
0007h:	10h	Загрузить
	20h	Сложить
	20h	Сложить
	11h	Сохранить
000Bh:	FFh	Остановить

Возможно, вам захочется сравнить содержимое этого массива RAM с массивом, в котором хранятся слагаемые. В результате вы заметите, что каждый код в массиве «Код» соответствует значению в массиве

«Данные», которое должно быть загружено в аккумулятор, прибавлено к его содержимому или сохранено в памяти. Используемые таким образом числовые коды часто называются *кодами команд* или *кодами операций*. Они дают схеме «команду» выполнить определенную «операцию».

Как я уже упоминал, выход 8-битной защелки исходного сумматора должен быть входом массива RAM «Данные». Так работает команда «Сохранить». Однако нам требуется внести еще одно изменение: изначально выход 8-битного сумматора — вход 8-битной защелки. Теперь для выполнения команды «Загрузить» выход массива «Данные» иногда должен соединяться со входом 8-битной защелки. Для этого необходим селектор двух линий на одну. Пересмотренная схема сумматора выглядит следующим образом.



На этой схеме еще чего-то не хватает, но она отображает все 8-битные потоки данных между различными компонентами. Шестнадцатибитный счетчик

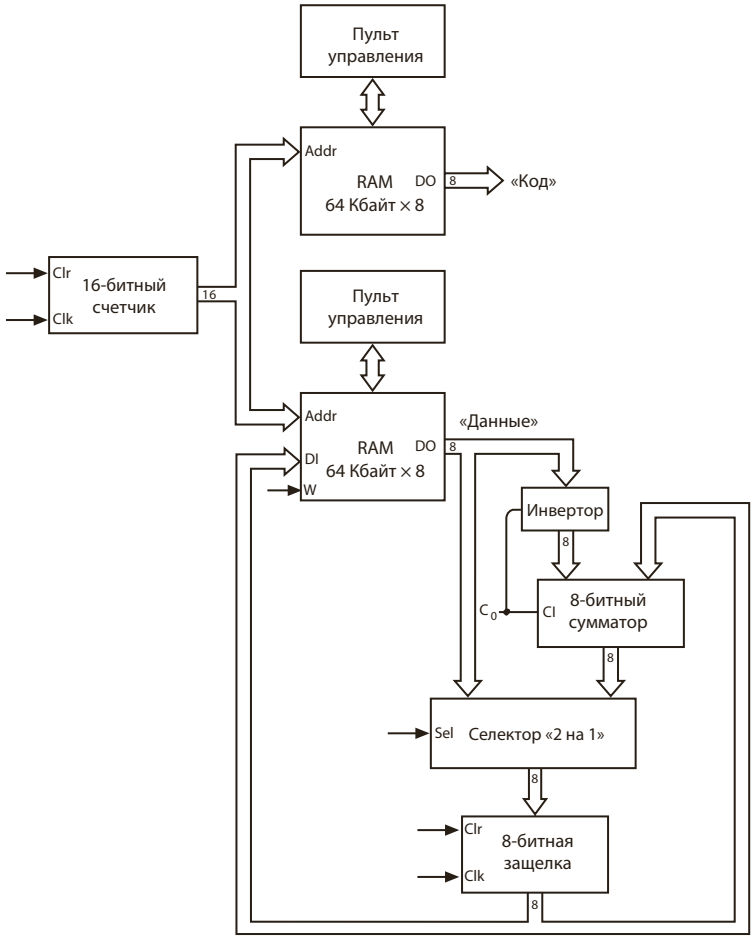
предоставляет адреса для двух массивов RAM. Выход массива «Данные» подключен ко входу 8-битного сумматора для выполнения команды «Сложить». Правда, ко входу 8-битной защелки может быть подключен либо выход массива «Данные» (в случае выполнения команды «Загрузить»), либо выход сумматора (в случае выполнения команды «Сложить»). В этой ситуации требуется селектор «2 на 1». Выходной сигнал защелки не только возвращается обратно в сумматор, но и подается на вход массива «Данные» для выполнения операции «Сохранить».

На схеме не хватает только контролирующих эти компоненты сигналов, которые называются *управляющими*; к ним относятся входы Clk и Clr 16-битного счетчика, входы Clk и Clr 8-битной защелки, вход W массива «Данные» и вход Sel селектора «2 на 1». Некоторые из этих сигналов, очевидно, будут основываться на выходе массива «Код». Например, вход Sel селектора «2 на 1» должен быть равен 0 (выбран выход «Данные» массива RAM), если выходной сигнал массива «Код» соответствует команде «Загрузить». Вход W массива «Данные» должен быть равен 1 только тогда, когда код соответствует команде «Сохранить». Эти управляющие сигналы могут генерироваться различными комбинациями логических вентилей.

Добавив несколько дополнительных компонентов и код новой команды, можем сделать так, чтобы наша схема вычитала число из значения, хранящегося в аккумуляторе. Первым делом нужно расширить таблицу кодов команд.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Остановить	FFh

Коды команд «Сложить» и «Вычесть» отличаются только младшим битом значения, который мы будем называть C_0. Если значение кода команды равно 21h, то схема должна делать то же самое, что и в случае выполнения команды «Сложить», за исключением того, что данные из массива «Данные» инвертируются перед попаданием в сумматор, а для входа сумматора CI задается значение 1. Сигнал C_0 может выполнять обе операции в обновленном сумматоре, дополненном инвертором.



Предположим, нам нужно сложить два числа: 56h и 2Ah, а затем из полученной суммы вычесть 38h. Это можно сделать, используя следующие коды и данные, хранящиеся в двух массивах RAM.

	«Код»			«Данные»	
0000h:	10h	Загрузить	0000h:	56h	
	20h	Сложить		2Ah	
	21h	Вычесть		38h	
	11h	Сохранить			← Результат сохраняется здесь
	FFh	Остановить			

После выполнения операции «Загрузить» аккумулятор содержит значение 56h, после операции «Сложить» — сумму 56h и 2Ah, то есть 80h. Операция «Вычесть» приводит к инвертированию битов следующего значения в массиве «Данные» (38h). Инвертированное значение C7h прибавляется к 80h, при этом вход сумматора для переноса (CI) равен 1.

```
  C7h
+ 80h
+  1h
-----
  48h
```

Результатом будет 48h (в десятичной системе счисления: 86 + 42 – 56 = 72).

Еще одной нерешенной проблемой остается недостаточная ширина канала данных сумматора и всех остальных подключенных к нему устройств. Ранее я предлагал удвоить количество 8-битных сумматоров (и всех остальных компонентов), чтобы получить 16-битные устройства.

Однако мы можем использовать гораздо менее дорогостоящее решение. Предположим, нужно сложить два 16-битных числа, например следующие.

```
  76ABh
+ 232Ch
-------
```

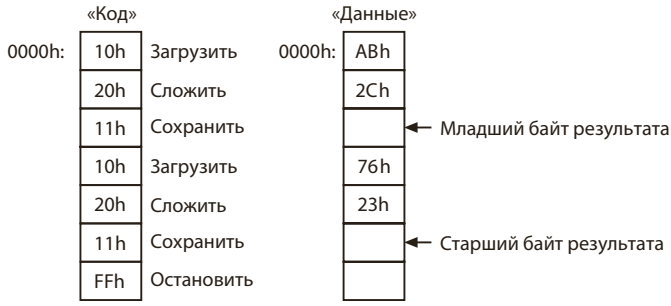
Для получения суммы двух 16-битных чисел отдельно сложим их *младшие* байты (крайние справа).

```
  ABh
+ 2Ch
-----
  D7h
```

А затем *старшие* (крайние слева).

```
  76h
+ 23h
-----
  99h
```

В результате получится число 99D7h. Если мы сохраним два 16-битных числа в памяти, как показано на рисунке, результат D7h будет сохранен по адресу 0002h, а 99h — по адресу 0005h.



Разумеется, это будет работать не всегда. Такой метод подходит для сложения чисел, выбранных в качестве примера. Если нам требуется сложить числа 76ABh и 236Ch, при сложении двух младших байтов возникает перенос.

$$\begin{array}{r} \text{ABh} \\ +\ \text{6Ch} \\ \hline \text{117h} \end{array}$$

Этот перенос должен быть прибавлен к сумме двух старших байтов для получения окончательного результата — 9A17h.

$$\begin{array}{r} \text{1h} \\ +\ \text{76h} \\ +\ \text{23h} \\ \hline \text{9Ah} \end{array}$$

Можем ли мы сделать так, чтобы наш сумматор корректно складывал два 16-битных числа? Да, для этого достаточно *сохранить бит переноса*, полученный от 8-битного сумматора при выполнении первой операции сложения, а затем подать этот бит на вход сумматора для переноса при следующем сложении. Как можно сохранить этот бит? С помощью однобитной защелки (на этот раз назовем ее *защелкой для переноса*).

Использование защелки для переноса требует еще одного кода команды — «Сложить с переносом». При сложении 8-битных чисел используется обычная команда «Сложить». На вход сумматора для переноса (CI) подается значение 0, а значение, возникающее на выходе для переноса (CO), сохраняется в защелке для переноса (хотя в его использовании вообще нет необходимости).

При суммировании двух 16-битных чисел используем обычную команду «Сложить» для сложения младших байтов. Вход сумматора CI равен 0, а значение

выхода CO сохраняется в защелке для переноса. Для сложения двух старших байтов будем использовать новую команду «Сложить с переносом». В данном случае при сложении двух чисел на вход сумматора CI подается значение, сохраненное в защелке для переноса. Таким образом, если в результате первой операции сложения возник перенос, этот бит переноса используется при втором сложении. Если переноса не возникло, выходное значение защелки для переноса равно 0.

Для вычитания одного 16-битного числа из другого потребуется еще одна новая команда — «Вычесть с заимствованием». Как правило, выполнение команды «Вычесть» предполагает инвертирование вычитаемого и подачу на вход сумматора CI значения 1. В этом случае выход для переноса также равен 1, — это нормально, на это явление можно не обращать внимания. Однако при вычитании 16-битного числа значение этого выхода необходимо сохранить в защелке для переноса. При втором вычитании это значение должно быть подано на вход сумматора CI.

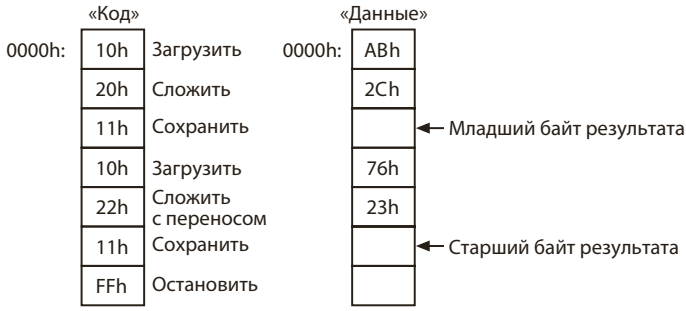
Учитывая две новые операции, «Сложить с переносом» и «Вычесть с заимствованием», в общей сложности мы имеем семь кодов команд.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Остановить	FFh

Число, подаваемое в сумматор, инвертируется при выполнении операции «Вычесть» или «Вычесть с заимствованием». Выходной сигнал сумматора CO подается на вход защелки для переноса. Он сохраняется в защелке всякий раз, когда выполняются операции «Сложить», «Вычесть», «Сложить с переносом» или «Вычесть с заимствованием». Значение входа для переноса 8-битного сумматора устанавливается в 1 при выполнении операций «Вычесть», или «Сложить с переносом», или «Вычесть с заимствованием», когда выход защелки для переноса равен 1.

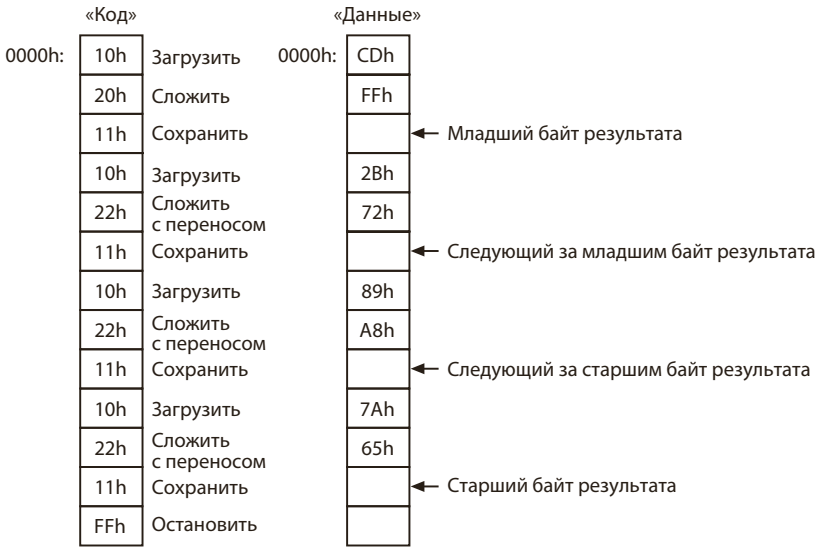
В результате выполнения команды «Сложить с переносом» значение входа сумматора для переноса бывает равно 1 только в том случае, когда при выполнении предыдущей команды «Сложить» или «Сложить с переносом» на выходе из сумматора возник перенос. Таким образом, мы используем команду «Сложить

с переносом» всякий раз, когда складываем многобайтные числа, вне зависимости от того, есть ли в этой операции необходимость. Вот как следует закодировать продемонстрированную ранее операцию сложения 16-битных чисел.



Такая последовательность операций позволяет получить правильный ответ независимо от того, какие цифры мы складываем.

Добавление двух новых кодов команд значительно расширило функционал сумматора. Мы больше не ограничиваемся сложением 8-битных значений. Многократное использование команды «Сложить с переносом» позволяет складывать 16-, 24-, 32-, 40-битные значения и т. д. Предположим, нам нужно сложить 32-битные числа 7A892BCDh и 65A872FFh. Для этого потребуется лишь одна команда «Сложить» и три команды «Сложить с переносом».



Конечно, вводить эти числа в память не очень удобно. При этом не только приходится использовать переключатели для представления двоичных чисел.

Сами числа сохраняются в несмежных ячейках, например 7A892BCDh оказывается в ячейках 0000h, 0003h, 0006h и 0009h, начиная с младшего байта. Для получения окончательного результата необходимо проверить значения, хранящиеся в ячейках 0002h, 0005h, 0008h и 000Bh.

Более того, текущая конструкция нашего сумматора не допускает повторного использования результатов в последующих операциях. Допустим, нужно сложить три 8-битных числа, а затем вычесть из этой суммы другое 8-битное число и сохранить результат. Для этого потребуются команда «Загрузить», две команды «Сложить», команды «Вычесть» и «Сохранить». А если нужно вычесть из этой исходной суммы другие числа, в то время как сумма эта недоступна, поскольку каждый раз нам пришлось бы ее пересчитывать? Проблема в том, что созданный нами сумматор обращается к ячейкам массивов «Код» и «Данные» одновременно и последовательно, начиная с адреса 0000h. Каждая команда в массиве «Код» соответствует ячейке массива «Данные» по тому же адресу. Когда в результате выполнения команды «Сохранить» в массиве «Данные» сохраняется некоторое значение, в дальнейшем оно уже не может быть загружено в аккумулятор.

Чтобы решить эту проблему, я намерен внести в сумматор фундаментальное и радикальное изменение, которое поначалу может показаться безумно сложным. Однако со временем (надеюсь) вы оцените ту гибкость, которую оно обеспечивает.

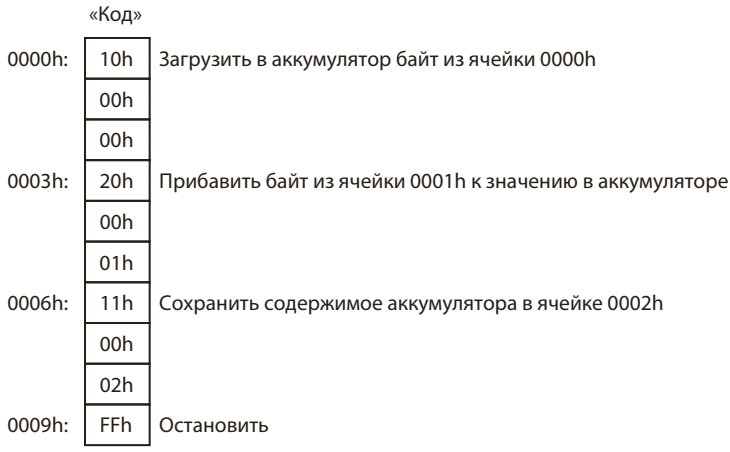
Итак, в настоящее время у нас есть семь кодов команд.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Остановить	FFh

Каждый из этих кодов занимает в памяти один байт. Теперь нужно, чтобы эти коды, за исключением кода команды «Остановить», занимали по три байта. Первый байт будет занят самим кодом, а в следующих двух будет храниться 16-битный адрес ячейки памяти. Для команды «Загрузить» этот адрес указывает местоположение в массиве «Данные», где содержится байт для загрузки в аккумулятор. Для команд «Сложить», «Вычесть», «Сложить с переносом» и «Вычесть с заимствованием» этот адрес указывает местоположение

байта, который должен быть прибавлен или вычтен из значения, содержащегося в аккумуляторе. Для команды «Сохранить» этот адрес указывает местоположение, где нужно сохранить содержимое аккумулятора. Например, простейшая задача для текущей версии сумматора — сложение двух чисел. Для этого в массивы «Код» и «Данные» необходимо внести следующие значения.

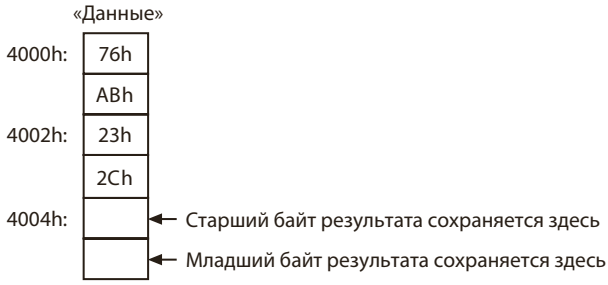
В пересмотренной версии сумматора для хранения каждой команды, кроме «Остановить», требуется три байта.



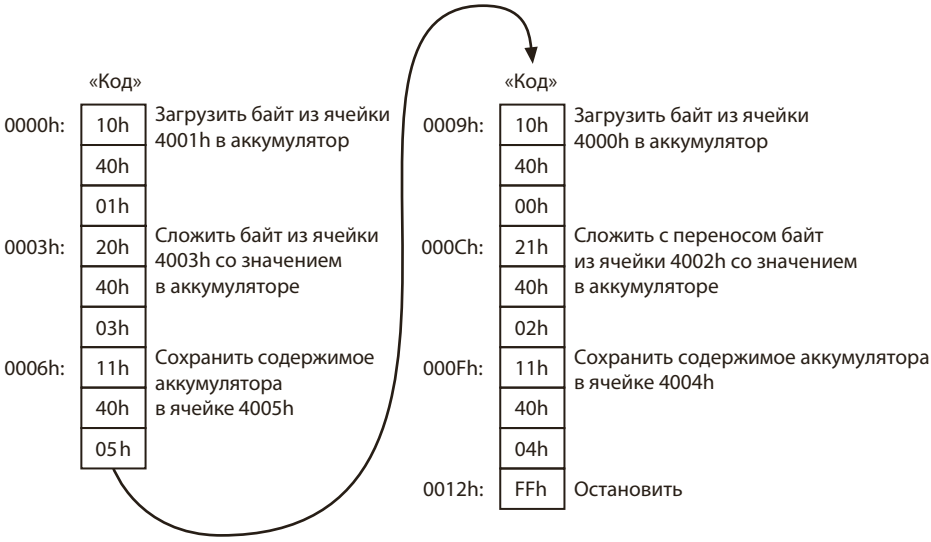
За каждым из кодов команд, кроме «Остановить», следуют два байта, указывающие 16-битный адрес в массиве «Данные». В данном примере этими тремя адресами являются 0000h, 0001h и 0002h, однако они могут быть какими угодно.

Ранее я продемонстрировал процесс сложения двух 16-битных чисел, в частности 76ABh и 232Ch, с использованием команд «Сложить» и «Сложить с переносом». Однако нам пришлось сохранить два младших байта этих чисел в ячейках 0000h и 0001h и два старших байта — в ячейках 0003h и 0004h. Результат сложения был сохранен в ячейках 0002h и 0005h.

Благодаря этому изменению мы можем хранить слагаемые и результат более разумным способом, возможно, в той области памяти, которую еще никогда не использовали.



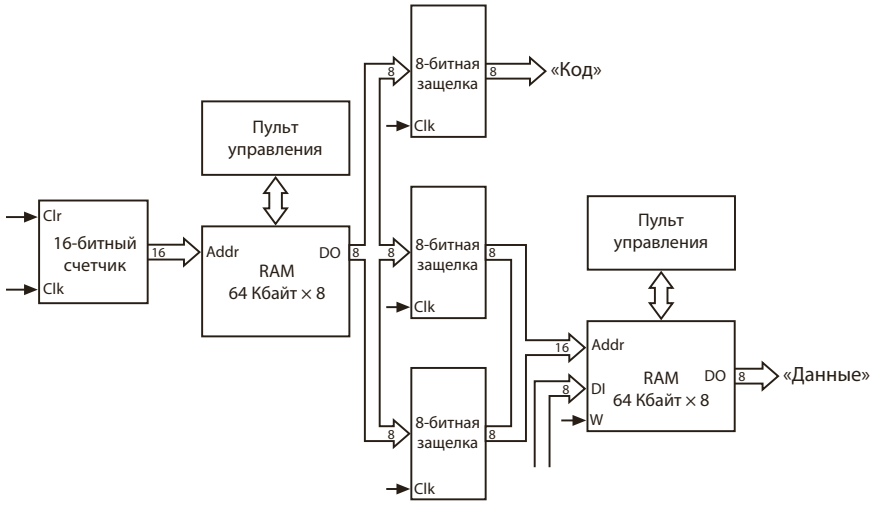
Эти шесть ячеек не обязательно должны располагаться последовательно. Они могут быть разбросаны по всему массиву «Данные». Для сложения чисел, находящихся в этих ячейках памяти, в массиве «Код» необходимо сохранить следующие значения.



Обратите внимание: сначала складываются два младших байта, хранящихся в ячейках 4001h и 4003h, а результат сохраняется в ячейке 4005h. Два старших байта (содержащихся в ячейках 4000h и 4002h) складываются с использованием команды «Сложить с переносом», а результат сохраняется в ячейке 4004h. Если удалим команду «Остановить» и добавим новые команды в массив «Код», то при выполнении последующих расчетов сможем использовать исходные слагаемые и их сумму путем простого обращения к соответствующим адресам.

Ключевым моментом при реализации этой идеи является подключение выхода массива «Код» DO к трем 8-битным защелкам. В каждой из этих защелок хранится один из байтов 3-байтной команды. Первая защелка содержит код команды, вторая — старший байт адреса, третья — младший байт адреса.

Выход второй и третьей защелок становится 16-битным адресом ячейки в массиве «Данные».



Процесс извлечения команды из памяти называется *выборкой команды*. Каждая команда в нашем сумматоре занимает три байта, и она извлекается из памяти по одному байту за раз; для извлечения команды требуются три цикла синхросигнала, для всего *командного цикла* — четыре. Эти изменения, безусловно, усложняют систему управляющих сигналов.

Когда сумматор производит серию действий в соответствии с кодом, это называется *выполнением команды*. Однако это не значит, что машина *живая*. Она не анализирует машинный код и не решает, что ей делать. Каждый машинный код просто особым образом запускает различные управляющие сигналы, заставляя машину выполнять различные действия.

Важно: расширение функционала сумматора приводит к замедлению его работы. При той же частоте осциллятора машина складывает числа в четыре раза медленнее по сравнению с первым сумматором, описанным в этой главе. Это проявление инженерного принципа «Бесплатных завтраков не бывает»*, смысл которого в том, что улучшение одного аспекта машины приводит к ухудшению другого.

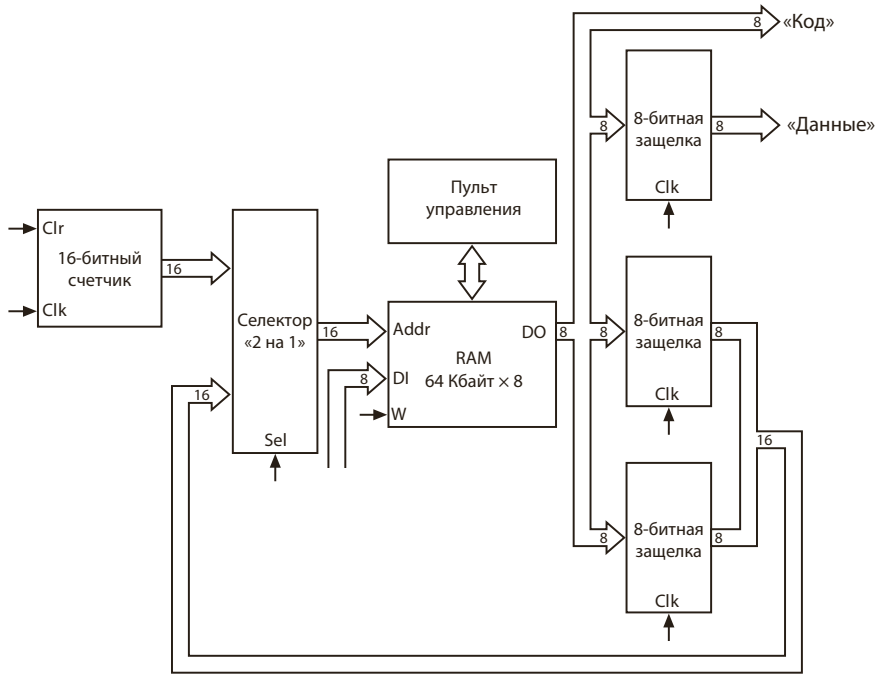
---

* В математическом фольклоре теорема No free lunch означает, что два любых алгоритма оптимизации эквивалентны, если их эффективность сравнивается посредством средних показателей по всем возможным проблемам. Так пояснили в 2005 году Дэвид Волперт и Вильям Макриди, в работе которых (No Free Lunch Theorems for Optimization) это словосочетание появилось впервые (в 1995 году). Сама работа носила более математический и формальный характер, а теоремы, предложенные в ней, были математически доказаны. *Прим. науч. ред.*

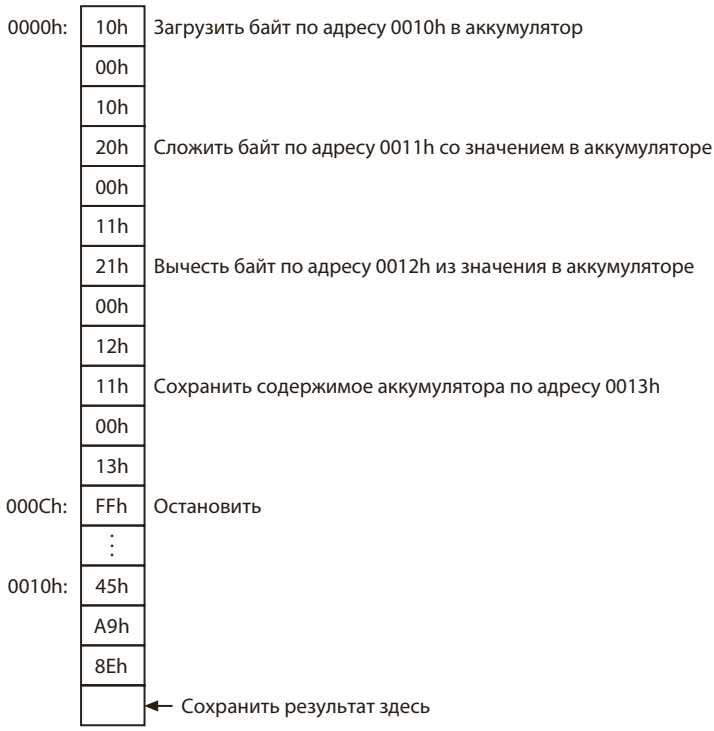
Если бы вы на самом деле собирали такой сумматор из реле, то основными компонентами схемы, очевидно, были бы два массива RAM емкостью 64 килобайта. В самом начале вы, вероятно, сэкономили на этих компонентах, решив, что пока можете обойтись одним килобайтом памяти. Если бы вы были уверены, что сохраните все данные в ячейках с 0000h по 03FFh, то могли бы спокойно использовать память емкостью менее 64 килобайта.

Впрочем, вы, вероятно, не в восторге от необходимости использовать *два* массива RAM. На самом деле это необязательно. Я ввел два массива RAM (один с кодами и один с данными) для того, чтобы архитектура сумматора была максимально понятной и простой. Но сейчас, когда мы решили, что каждая команда будет занимать три байта (причем второй и третий байты будут содержать адрес, где находятся данные), нам не нужны два отдельных массива RAM. Коды команд и данные можно хранить в одном массиве.

Чтобы реализовать это, нужен селектор «2 на 1» для определения способа адресации массива RAM. Как правило, один адрес, как и раньше, подается на вход селектора с 16-битного счетчика. Выход DO массива RAM по-прежнему подключен к трем защелкам, в которых сохраняются код команды и два байта адреса, сопровождающего каждую команду. Однако 16-битный адрес подается и на второй вход селектора «2 на 1». После сохранения этого адреса в защелках селектор передает его на адресный вход массива RAM.



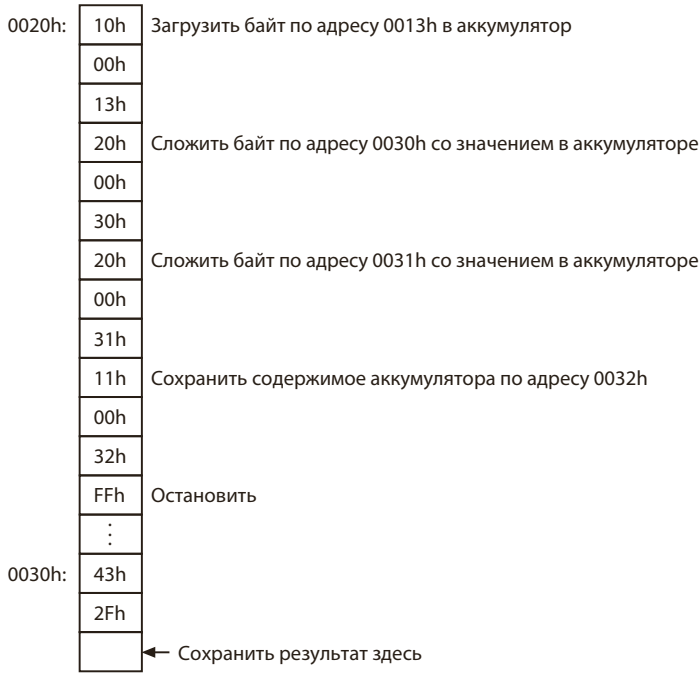
Мы значительно продвинулись. Теперь можно ввести команды и данные в один массив RAM. Например, на следующей диаграмме показан процесс сложения двух 8-битных чисел и вычитания из полученной суммы третьего числа.



Как обычно, команды сохраняются начиная с адреса 0000h, поскольку именно с этой ячейки счетчик начинает адресацию массива RAM после обнуления. Последняя команда «Сохранить» хранится по адресу 000Ch. Мы могли бы сохранить три числа и результаты в любом месте массива RAM (разумеется, за исключением первых 13 байт, поскольку они заняты кодами команд), однако решили остановиться на данных, начиная с ячейки по адресу 0010h.

Предположим, что нам понадобилось прибавить к результату еще два числа. Можно, конечно, заменить все только что введенные команды новыми, но мы не хотим этого делать. Возможно, мы предпочли бы после выполнения этих команд просто выполнить новые, заменив перед этим код команды «Остановить» кодом новой команды «Загрузить» в ячейке 000Ch. Однако нам требуются две новые команды «Сложить», команда «Сохранить» и новая команда «Остановить». Единственная проблема: в ячейке 0010h хранятся данные. Их необходимо переместить дальше, изменив при этом ссылающиеся на них команды.

Может показаться, что объединение кода и данных в одном массиве RAM не было такой уж хорошей идеей. Уверяю вас, рано или поздно такая проблема обязательно бы возникла. Так что давайте решим ее. В данном случае можно попробовать ввести коды новых команд, начиная с адреса 0020h, а новые данные — с адреса 0030h.



Обратите внимание: первая команда «Загрузить» ссылается на ячейку 0013h, в которой хранится результат первого расчета.

Итак, с адреса 0000h в памяти хранятся команды, с 0010h — некоторые данные, с 0020h — еще команды, а с адреса 0030h — еще данные. Нам нужно, чтобы сумматор выполнил все команды, начиная с адреса 0000h.

Мы знаем, что следует удалить команду «Остановить» из ячейки 000Ch. Под словом «*удалить*» я подразумеваю ее замену чем-то другим. Достаточно ли этого?

Проблема в том, что все, чем мы заменим команду «Остановить», будет интерпретироваться как код команды. Это касается и того, что будет храниться через каждые три ячейки после него — по адресам 000Fh, 0012h, 0015h, 0018h, 001Bh и 001Eh. Что, если одним из этих значений окажется число 11h, которое соответствует команде «Сохранить»? Что, если два байта после кода команды «Сохранить» будут ссылаться на ячейку 0023h? Это заставит сумматор сохранить содержимое аккумулятора в этой ячейке. Однако в ней уже содержится

что-то важное! И даже если ничего подобного не произойдет, после кода команды по адресу 001Eh сумматор извлечет код из ячейки 0021h, а не 0020h, где на самом деле находится код нашей следующей команды.

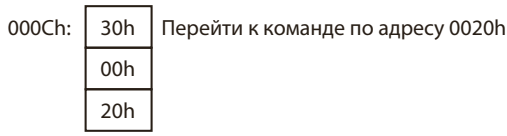
Все ли согласны, что мы не можем просто удалить команду «Остановить» из ячейки 000Ch и надеяться на лучшее?

Мы *можем* заменить ее новой командой под названием «Перейти». Давайте добавим ее в наш репертуар.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Перейти	30h
Остановить	FFh

Обычно массив RAM в сумматоре адресуется последовательно. Команда «Перейти» заставляет машину действовать иначе, то есть обращаться к ячейке массива по другому заданному адресу. Такая команда иногда называется командой *ветвления*.

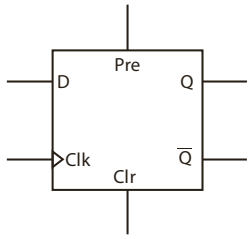
В предыдущем примере мы можем заменить команду «Остановить» в ячейке 000Ch командой «Перейти».



Значение 30h соответствует коду команды «Перейти». Шестнадцатибитный адрес, который следует за ним, указывает на ячейку со следующей командой, к которой должен обратиться сумматор.

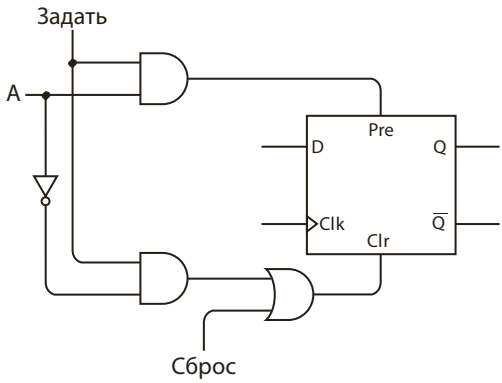
В предыдущем примере сумматор, как обычно, начинает с ячейки 0000h, обрабатывает команды «Загрузить», «Сложить», «Вычесть» и «Сохранить». Затем он выполняет команду «Перейти» и продолжает работу с адреса 0020h, начиная с которого хранятся команда «Загрузить», две команды «Сложить», команды «Сохранить» и «Остановить».

Команда «Перейти» влияет на 16-битный счетчик. Всякий раз, когда сумматор ее запускает, на выходе счетчика должен возникать новый адрес, следующий за кодом команды «Перейти». Это реализуется с помощью входов Pre и Clr триггеров D-типа со срабатыванием по фронту, из которых состоит 16-битный счетчик.



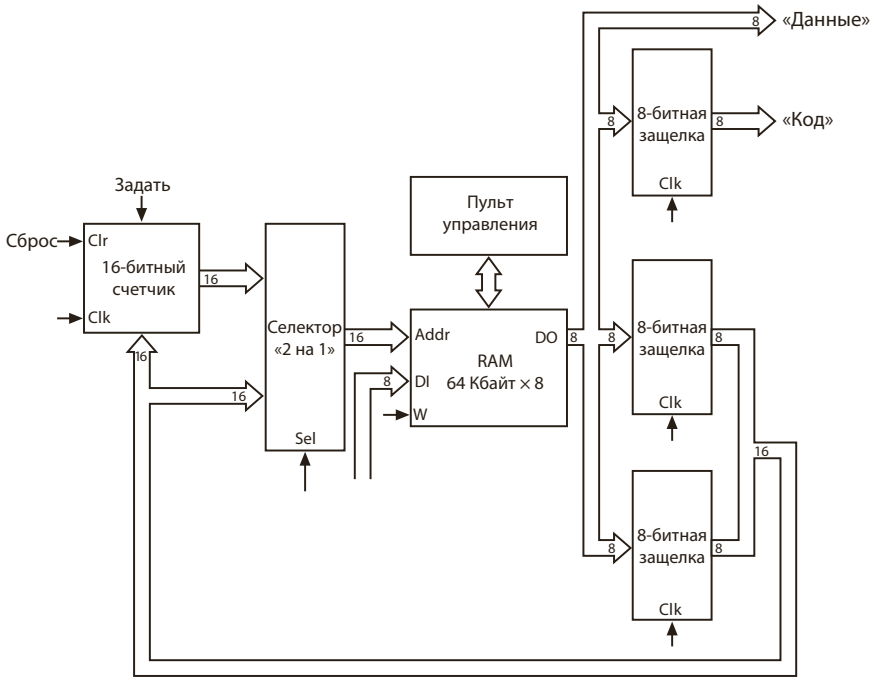
Напомню, что входы Pre и Clr должны быть равны 0 при выполнении обычной операции. Если вход Pre равен 1, сигнал Q тоже становится 1; если вход Clr равен 1, сигнал Q — 0.

Если хотите загрузить в триггер новое значение (назовем его A — от слова «адрес»), вы можете включить его в схему следующим образом.



Обычно сигнал «Задать» равен 0. В этом случае вход триггера Pre также 0. Вход Clr равен 0 при условии, что сигнал «Сброс» не равен 1. Это позволяет очистить триггер независимо от значения сигнала «Задать». Когда сигнал «Задать» — 1, вход Pre — 1, а вход Clr — 0 при условии, что сигнал A — 1. Если сигнал A — 0, то вход Pre — 0, а вход Clr — 1. Следовательно, значение выхода Q будет совпадать со значением A.

Нам требуется по одной такой схеме для каждого бита 16-битного счетчика. После загрузки конкретного значения счетчик будет продолжать подсчет с него. В других отношениях изменения не являются такими серьезными. Шестнадцатибитный адрес из массива RAM, сохраненный в защелках, подается как на вход селектора «2 на 1» (передает его на адресный вход массива RAM), так и на вход 16-битного счетчика для выполнения функции «Задать».



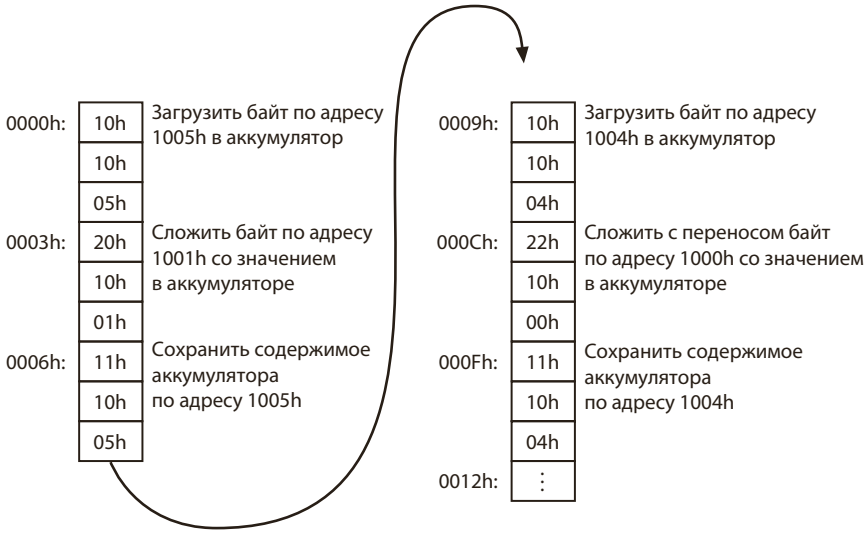
Очевидно, сигнал «Задать» должен быть равен 1, только если код команды 30h и адрес сохранен в защелках.

Команда «Перейти», безусловно, полезна. Однако еще более практичной была бы команда, которая совершает переход не всегда, а только при определенных условиях. Такая команда называется *условным переходом*. Вероятно, лучший способ продемонстрировать ее полезность — постановка вопроса: «Как сделать так, чтобы наш сумматор *перемножил* два 8-битных числа, например A7h и 1Ch?»

Все просто, не так ли? Результат умножения двух 8-битных значений — 16-битное число. Для удобства все три числа, участвующие в этой операции, выражены в виде 16-битных значений. Первым делом нужно решить, где следует сохранить эти числа и результат.

1000h:	00h	16-битный множитель
	A7h	
1002h:	00h	16-битное множимое
	1Ch	
1004h:	00h	← 16-битное произведение сохраняется здесь…
	00h	← …и здесь

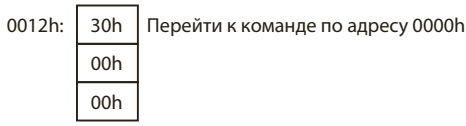
Все знают, что умножить число A7h на 1Ch (соответствует десятичному числу 28) — это то же самое, что найти сумму 28 чисел A7h. Таким образом, в ячейках 1004h и 1005h фактически будет накапливаться 16-битный результат этого суммирования. Вот последовательность кодов для выполнения первой операции сложения.



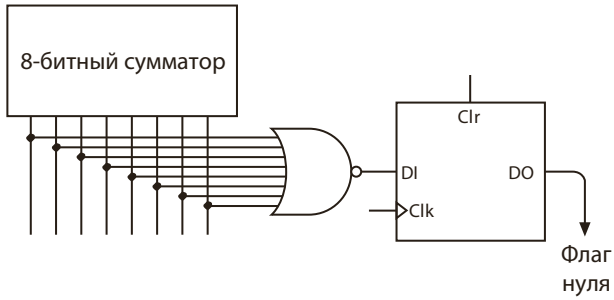
После выполнения этих шести команд 16-битное значение в ячейках 1004h и 1005h будет равно числу A7h, умноженному на 1. Чтобы это значение равнялось произведению A7h и 1Ch, эти шесть команд необходимо выполнить еще 27 раз. Вы можете ввести эти шесть команд еще 27 раз, начиная с адреса 0012h, или сохранить код команды «Остановить» по адресу 0012h, нажать кнопку «Сброс» 28 раз, чтобы получить окончательный ответ.

Конечно, ни один из этих двух вариантов не идеален. Оба они предполагают, что вы должны что-то сделать: ввести набор команд или нажать кнопку «Сброс» в соответствии со значением одного из сомножителей. Уверен, что вы не хотели бы всю жизнь перемножать 16-битные значения именно так.

А что, если поместить команду «Перейти» в ячейку 0012h? Эта команда заставляет счетчик снова начать счет с ячейки 0000h.



Вроде бы проблема решена. При выполнении первой команды 16-битное значение в ячейках 1004h и 1005h будет равно произведению чисел A7h и 1. Затем благодаря команде «Перейти» мы вернемся к началу. После выполнения второй операции 16-битный результат будет равен произведению чисел A7h и 2. В конце концов он станет равен произведению чисел A7h и 1Ch, однако у нас нет способа остановить работу сумматора. Нам нужна команда «Перейти», которая запускала бы процесс с начала столько раз, сколько необходимо. Это условный переход, который совсем несложно реализовать. Сначала нужно добавить однобитную защелку, подобную защелке для переноса. Она будет называться *нулевой защелкой*, поскольку в ней будет сохраняться значение 1, только если все выходы 8-битного сумматора будут равны 0.



Выход этого 8-битного вентиля ИЛИ-НЕ равен 1, только если все входы равны 0. Как и вход защелки для переноса Clk, вход Clk нулевой защелки сохраняет значение только при выполнении команд «Сложить», «Вычесть», «Сложить с переносом» или «Вычесть с заимствованием». Это сохраненное значение называется *флагом нуля*. Будьте внимательны: флаг нуля равен 1, если все выходы сумматора равны 0; флаг нуля — 0, если не все выходы сумматора равны 0.

Защелка для переноса и нулевая защелка позволяют дополнить наш репертуар четырьмя командами.

Операция	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Перейти	30h
Перейти, если 0	31h

Операция	Код
Перейти, если перенос	32h
Перейти, если не 0	33h
Перейти, если не перенос	34h
Остановить	FFh

Например, при выполнении команды «Перейти, если не 0» переход к указанному адресу осуществляется только в том случае, если выход нулевой защелки равен 0. Другими словами, никакого перехода не произойдет, если результат выполнения последней команды — «Сложить», «Вычесть», «Сложить с переносом» или «Вычесть с заимствованием» — равен 0. Реализация этого нововведения предполагает дополнение набора управляющих сигналов, которые реализуют обычную команду «Перейти»: при выполнении команды «Перейти, если не 0» сигнал 16-битного счетчика «Задать» равен 1 только при нулевом значении флага нуля.

Следующая последовательность команд, начинающаяся с адреса 0012h, позволяет перемножить два числа.

0012h:	10h	Загрузить байт по адресу 1003h в аккумулятор
	10h	
	03h	
0015h:	20h	Сложить байт по адресу 001Eh со значением в аккумуляторе
	00h	
	1Eh	
0018h:	11h	Сохранить содержимое аккумулятора по адресу 1003h
	10h	
	03h	
001Bh:	33h	Перейти к ячейке 0000h, если флаг нуля не равен 1
	00h	
	00h	
001Eh:	FFh	Остановить

При первом выполнении этой последовательности команд в ячейках 0004h и 0005h, как мы уже выяснили, хранится 16-битное произведение чисел A7h и 1. Приведенные здесь команды загружают байт из ячейки 1003h в аккумулятор. Его значение — 1Ch. Этот байт прибавляется к значению в ячейке 001Eh, в которой хранится команда «Остановить», представленная действительным числом. Сложение чисел FFh и 1Ch равнозначно вычитанию 1 из числа 1Ch, поэтому результат равен 1Bh. Это значение не равно 0, поэтому флаг нуля — 0.

Байт 1Bh сохраняется в ячейке 1003h. Затем выполняется команда «Перейти, если не 0». Флаг нуля не равен 1, поэтому переход происходит. На следующем этапе запускается команда, хранящаяся в ячейке 0000h. Имейте в виду, что команда «Сохранить» не влияет на флаг нуля. На его значение воздействуют только команды «Сложить», «Вычесть», «Сложить с переносом» или «Вычесть с заимствованием», поэтому его значение останется таким, какое было установлено в последний раз, когда отрабатывалась одна из этих команд.

При втором выполнении последовательности команд в ячейках 1004h и 1005h будет содержаться 16-битное произведение чисел A7h и 2. При сложении 1Bh и FFh получается 1Ah. Это значение не равно 0, поэтому мы возвращаемся к началу.

При двадцать восьмом выполнении этой последовательности команд в ячейках 1004h и 1005h будет содержаться произведение чисел A7h и 1Ch. В ячейке 1003h будет храниться значение 1, которое прибавится к FFh, в результате чего получится 0. При этом значение флага нуля станет 1, поэтому вместо команды «Перейти, если 0» будет выполнена команда «Остановить».

Теперь я могу объявить, что нам наконец удалось собрать устройство, которое мы можем на полном основании назвать *компьютером*. Несмотря на примитивность, это уже компьютер. Его отличие заключается в условном переходе. Контролируемое повторение последовательности команд, или *цикл,* — это то, что отличает компьютеры от калькуляторов. Я только что продемонстрировал, как команда условного перехода позволяет машине перемножить два числа. Подобным образом она может выполнить и деление. Кроме того, она не ограничивается 8-битными значениями и способна складывать, вычитать, умножать и делить 16-, 24-, 32-битные числа и числа большей разрядности. Это значит, что такая машина может вычислять квадратные корни, логарифмы и тригонометрические функции.

Когда мы собрали компьютер, можем начать использовать соответствующую терминологию.

Собранный компьютер можно классифицировать как *цифровой*, поскольку он работает с дискретными числами. К другому типу относятся *аналоговые* компьютеры, которые в настоящее время уже практически не применяются. (Цифровые, или *дискретные,* данные могут принимать только определенные значения*. Аналоговые данные непрерывны и могут принимать любое значение в заданном диапазоне.)

---

* В строгом смысле понятия «дискретный» и «цифровой» не эквивалентны. Дискретный сигнал представляет собой последовательность отсчетов (например, напряжения) с произвольными значениями. Цифровым сигнал станет после того, как будет *проквантован* (то есть значения отсчетов будут округлены до ближайшего разрешенного значения) и закодирован с помощью нулей и единиц. *Прим. науч. ред.*

Цифровой компьютер состоит из четырех основных частей: *процессора*, *памяти*, как минимум одного *устройства ввода* и одного *устройства вывода*. В нашей машине память — массив RAM емкостью 64 килобайт. Устройства ввода и вывода — ряд переключателей и лампочек на пульте управления массивом RAM. Эти переключатели и лампочки позволяют нам (людям) вводить в память числа и анализировать результаты.

Все остальное — процессор, который также называется *центральным процессорным устройством (ЦПУ)*. В обычной речи процессор иногда называют *мозгом* компьютера, но я не хотел бы использовать этот термин, поскольку собранное нами устройство совершенно не похоже на мозг. В наши дни часто употребляется слово «*микропроцессор*» — процессор, который благодаря использованию технологии, которую я опишу в главе 18, отличается очень малыми размерами. К тому, что мы собрали из реле в этой главе, вряд ли применима приставка «*микро-*».

Созданный нами процессор является *8-битным*. Ширина аккумулятора и большинства каналов данных составляет 8 бит. Шестнадцатибитное значение подается только на адресный вход массива RAM. Если бы значение адреса было 8-битным, то вместо 65 536 байт можно было бы адресовать только 256 байт памяти, что было бы значительным ограничением.

Процессор состоит из нескольких компонентов. Я уже упомянул *аккумулятор*, который представляет собой просто защелку, позволяющую хранить число внутри процессора. Восьмибитный инвертор и 8-битный сумматор нашего компьютера вместе образуют *арифметико-логическое устройство (АЛУ)*. Наше АЛУ выполняет только арифметические операции, в частности сложение и вычитание. В более сложных компьютерах АЛУ может производить такие логические функции, как И, ИЛИ и исключающее ИЛИ. Шестнадцатибитный счетчик называется *счетчиком команд*.

Собранный нами компьютер состоит из реле, проводов, переключателей и лампочек. Все это — *аппаратное обеспечение* (hardware, или хард) компьютера. Напротив, содержащиеся в памяти коды команд и числа называются *программным обеспечением* (software, или софт).

Когда мы говорим о компьютерах, термин «*программное обеспечение*» практически идентичен термину «*компьютерная программа*» (или просто «*программа*»). Написание программного обеспечения называется *программированием*. Именно этим я занимался, когда определял серию команд, позволяющих нашему компьютеру перемножить два числа.

Как правило, в компьютерных программах мы можем различать *код* (который относится к самим командам) и *данные*, то есть числа, которыми манипулирует код. Иногда это различие не так очевидно. Например, команда

«Остановить» в описанной выше последовательности выступала еще и в качестве числа –1.

Компьютерное программирование иногда называют *написанием кода,* или *кодированием*. От программиста вы можете услышать фразы типа: «Я потратил свои каникулы на кодирование», «Я писал код до семи утра». Иногда компьютерных программистов называют *кодерами*, хотя кому-то этот термин может показаться уничижительным. Такие программисты, возможно, предпочтут, чтобы их называли *программными инженерами*.

Коды команд, на которые реагирует процессор (например, 10h и 11h, соответствующие командам «Загрузить» и «Сохранить»), называются *машинными кодами,* или *машинным языком*. В данном случае термин «язык» используется потому, что он напоминает устный или письменный человеческий язык в том смысле, что машина его «понимает» и реагирует на него.

Для обозначения команд, выполняемых нашей машиной, я использовал довольно длинные фразы вроде «Сложить с переносом». Как правило, команды — это короткие мнемонические коды, написанные заглавными буквами. Мнемокоды могут состоять всего из двух или трех букв. Далее перечислены возможные мнемокоды для команд, распознаваемых нашим компьютером.

Операция	**Код**	**Мнемокод**
Загрузить	10h	LOD
Сохранить	11h	STO
Сложить	20h	ADD
Вычесть	21h	SUB
Сложить с переносом	22h	ADC
Вычесть с заимствованием	23h	SBB
Перейти	30h	JMP
Перейти, если 0	31h	JZ
Перейти, если перенос	32h	JC
Перейти, если не 0	33h	JNZ
Перейти, если не перенос	34h	JNC
Остановить	FFh	HLT

Эти мнемокоды особенно полезны, когда используются в сочетании с другими сокращенными обозначениями. Например, вместо длинной команды «Загрузить байт из ячейки 1003h в аккумулятор» можем написать следующее.

```
LOD A,[1003h]
```

Обозначения A и [1003h] справа от мнемокода называются *аргументами*, которые определяют, что конкретно происходит при выполнении команды «Загрузить». При записи аргументов *место назначения* указывается слева (A — аккумулятор), а *источник* — справа. Квадратные скобки означают, что в аккумулятор нужно загрузить не число 1003h, а значение, хранящееся в ячейке памяти по адресу 1003h.

Аналогично команду «Сложить байт из ячейки 001Eh со значением в аккумуляторе» можно сократить до такого выражения.

```
ADD A, [001Eh]
```

А команду «Сохранить содержимое аккумулятора по адресу 1003h» — до выражения следующего вида.

```
STO [1003h], A
```

Обратите внимание: место назначения (ячейка памяти для команды «Сохранить») по-прежнему указывается слева, а источник — справа. Содержимое аккумулятора нужно сохранить в ячейке 1003h. Длинную команду «Перейти к ячейке 0000h, если флаг нуля не равен 1» можно записать кратко.

```
JNZ 0000h
```

В этой команде квадратные скобки не используются, поскольку она предполагает переход к ячейке 0000h, а не к значению, которое может в ней храниться.

Такая сокращенная запись удобна, поскольку позволяет перечислить команды в столбик и сделать их легко читаемыми, не прибегая к рисованию ячеек памяти. Для указания ячейки, в которой хранится определенная команда, можете использовать шестнадцатеричный адрес с двоеточием.

```
0000h: LOD A, [1005h]
```

А вот так можно указать на данные, хранящиеся по определенному адресу.

```
1000h: 00h, A7h
1002h: 00h, 1Ch
1004h: 00h, 00h
```

Два байта, разделенные запятой, указывают на то, что первый хранится в ячейке по адресу слева, а второй — в ячейке, следующей за ней. Эти три строки эквивалентны следующей строке.

```
1000h: 00h, A7h, 00h, 1Ch, 00h, 00h
```

Таким образом, программа для умножения двух чисел может быть записана в виде следующей серии команд.

```
0000h:  LOD A, [1005h]
ADD A, [1001h]
STO [1005h], A

LOD A, [1004h]
ADC A, [1000h]
STO [1004h], A

LOD A, [1003h]
ADD A, [001Eh]
STO [1003h], A

JNZ 0000h

001Eh:  HLT

1000h:  00h, A7h
1002h:  00h, 1Ch
1004h:  00h, 00h
```

Пустые строки и другие пробелы применяются для упрощения восприятия. При написании кода лучше не использовать фактические адреса, поскольку они могут измениться. Например, если вы решили сохранить числа в ячейках с 2000h до 20005h, то придется переписывать множество выражений. Лучше использовать *метки* для обозначения ячеек памяти. Эти метки — просто слова или то, что их напоминает.

```
BEGIN:  LOD A,[RESULT + 1]
        ADD A,[NUM1 + 1]
        STO [RESULT + 1], A
```

```
          LOD A,[RESULT]
          ADC A,[NUM1]
          STO [RESULT], A

          LOD A,[NUM2 + 1]
          ADD A,[NEG1]
          STO [NUM2 + 1], A

          JNZ BEGIN

NEG1:     HLT

NUM1:     00h, A7h
NUM2:     00h, 1Ch
RESULT:   00h, 00h
```

Важно: метки NUM1, NUM2 и RESULT ссылаются на ячейки памяти, где хранятся два байта. В приведенных выражениях метки NUM1 + 1, NUM2 + 1 и RESULT + 1 ссылаются на второй байт соответствующей метки. Обратите внимание на метку NEG1 (negative one, то есть «минус один») у команды HLT.

Наконец, чтобы не забыть, в чем заключается смысл того или иного выражения, вы можете добавить небольшие *комментарии* на естественном языке, отделив их от кода точкой с запятой.

```
BEGIN:    LOD A,[RESULT + 1]
          ADD A,[NUM1 + 1]; Прибавить младший байт
          STO [RESULT + 1], A

          LOD A,[RESULT]
          ADC A,[NUM1]; Прибавить старший байт
          STO [RESULT], A

          LOD A,[NUM2 + 1]
          ADD A,[NEG1]; Уменьшить второе число на 1
          STO [NUM2 + 1], A

          JNZ BEGIN

NEG1:     HLT
```

```
NUM1:   00h, A7h
NUM2:   00h, 1Ch
RESULT: 00h, 00h
```

Продемонстрированный язык компьютерного программирования называется *языком ассемблера*. Этот язык — некий компромисс между машинным кодом, состоящим исключительно из цифр, и многословными описаниями команд на человеческом языке, сопровождаемый адресами ячеек, выраженными в символьной форме. Люди иногда путают машинный код с языком ассемблера, поскольку это просто два разных способа выражения одного и того же. Каждый оператор на языке ассемблера соответствует определенным байтам машинного кода.

Если бы вы решили создать программу для собранного в этой главе компьютера, вероятно, сначала нужно было бы записать ее на бумаге на языке ассемблера. Когда сочтете, что программа написана правильно и готова к тестированию, вы вручную преобразуете каждый фрагмент кода на ассемблере в машинный код. После этого вы сможете использовать переключатели, чтобы ввести машинный код в массив RAM и *запустить программу*, то есть позволить ей выполнить набор введенных команд.

Изучение концепций компьютерного программирования предполагает знакомство с понятием *ошибки*. При кодировании, особенно при написании машинного кода, легко ошибиться. Ввод неправильного числа уже гарантирует возникновение неприятностей, но что произойдет, если неправильно ввести код команды? Если вы введете значение 11h (команда «Сохранить») вместо 10h (команда «Загрузить»), то машина не только не загрузит число, которое должна, но само это число будет заменено значением, которое в настоящий момент хранится в аккумуляторе. Некоторые ошибки могут приводить к непредсказуемым результатам. Предположим, вы используете команду «Перейти» для перехода к ячейке, которая не содержит действительного кода команды. Или, допустим, вы случайно применили команду «Сохранить», записав число в ячейке с кодом команды. Все может случиться (и частенько случается).

Даже в моей программе для умножения есть ошибка. Если вы запустите ее дважды, то при втором выполнении машина умножит A7h на 256 и прибавит произведение к уже полученному результату. Это связано с тем, что после первого выполнения программы в ячейке 1003h будет находиться значение 0. Когда вы запустите программу во второй раз, к этому значению будет прибавлено число FFh. Значение в ячейке 1003h будет отлично от 0, поэтому программа будет продолжать работу до тех пор, пока значение в этой ячейке не станет равным 0.

Наша машина может выполнять умножение, однако аналогичным образом она может выполнять и деление. Ранее я утверждал, что такая машина может использовать эти примитивные функции для вычисления квадратных корней, логарифмов и тригонометрических функций. Все, что ей требуется, — это оборудование для сложения и вычитания, а также способ реализации условного перехода для выполнения нужного кода. Любой программист скажет, что все остальное можно сделать с помощью программного обеспечения.

Разумеется, это программное обеспечение может быть довольно сложным. Существуют целые книги с описанием *алгоритмов*, используемых программистами для решения конкретных задач. К обсуждению этой темы мы пока не готовы. Ранее мы работали с целыми числами и не касались вопроса представления десятичных дробей в компьютере. Об этом поговорим в главе 23.

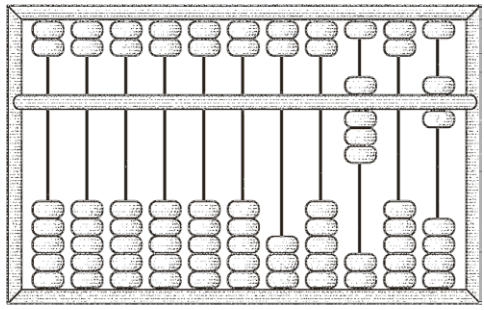
Я уже несколько раз упоминал, что все компоненты для создания таких устройств появились более ста лет назад. Однако компьютер, описанный в этой главе, вряд ли мог быть собран в то время. Многие из концепций, использованных в его конструкции, не были очевидны и в середине 1930-х годов, когда разрабатывались первые релейные компьютеры. Их начали осознавать примерно в 1945 году. До этого люди все еще пытались создавать компьютеры на основе десятичных, а не двоичных чисел. Кроме того, компьютерные программы не всегда хранились в памяти — иногда они были закодированы на бумажной ленте. На заре компьютерной эры память была очень дорогой и громоздкой. Создание массива RAM емкостью 64 килобайт из пяти миллионов телеграфных реле казалось такой же абсурдной идеей, как и сейчас.

Пришло время рассмотреть все, что мы узнали, в контексте истории развития вычислений и вычислительной техники. Возможно, мы обнаружим, что нам не придется собирать этот сложный релейный компьютер. Как я упоминал в главе 12, на смену реле пришли такие электронные устройства, как вакуумные лампы и транзисторы. Вероятно, мы также найдем, что кто-то уже создал устройство, эквивалентное нашему процессору и памяти, умещающееся на ладони.

Глава 18
От счетов к микросхемам

На протяжении всей записанной истории люди изобретали различные умные устройства и машины, стремясь хоть немного упростить процесс математических вычислений. Несмотря на то что человеческий вид, по-видимому, обладает врожденными способностями к вычислению, в этом нам часто требуется помощь. Нередко мы ставим такие сложные задачи, с которыми не можем справиться самостоятельно.

Развитие систем счисления можно считать самым ранним инструментом, помогавшим людям вести учет товаров и имущества. Представители многих культур, в том числе древние греки и американские индейцы, по-видимому, использовали для счета мелкие камешки и зерна. В Европе это привело к изобретению счетных досок, на востоке — счетов.



Несмотря на то что счеты обычно ассоциируются с азиатскими культурами, они, по-видимому, были завезены торговцами в Китай примерно в 1200 году н. э.

Никто никогда по-настоящему не получал удовольствия от умножения и деления, однако мало кто предпринимал какие-либо действия для решения этой проблемы. Шотландский математик Джон Непер (1550–1617) был одним из таких людей. Он изобрел логарифмы для упрощения счетных операций. Произведение двух чисел — это сумма их логарифмов. Так что, если нужно перемножить два числа, вы находите их в таблице логарифмов, складываете

числа из таблицы, а затем ищете в таблице число, логарифм которого соответствует полученной сумме. Построение таблиц логарифмов на протяжении последующих 400 лет занимало одни величайшие умы, в то время как другие разрабатывали небольшие устройства, заменяющие такие таблицы. Долгая история логарифмической линейки началась со счетной линейки, созданной Эдмундом Гюнтером (1581–1626) и усовершенствованной Уильямом Отредом (1574–1660). История этой линейки практически завершилась в 1976 году, когда компания Keuffel & Esser презентовала последнюю произведенную линейку Смитсоновскому институту в Вашингтоне (округ Колумбия). Причиной ее заката послужило изобретение ручного калькулятора.

Кроме того, Непер изобрел еще одно счетное устройство для облегчения умножения, состоящее из рядов чисел, выгравированных на кости или роге, благодаря чему оно получило название «кости Непера». Самый первый механический калькулятор в какой-то мере представлял автоматизированную версию костей Непера, созданную примерно в 1620 году Вильгельмом Шиккардом (1592–1635). Другие калькуляторы, сконструированные из сцепляющихся шестеренок и рычагов, являются почти такими же старыми приспособлениями. Среди наиболее выдающихся изобретателей механических калькуляторов можно выделить двух математиков и философов: Блеза Паскаля (1623–1662) и Готфрида Вильгельма Лейбница (1646–1716).

Наверняка вы помните, какую сложность создавал перенос бита как в первоначальном 8-битном сумматоре, так и в компьютере, который, помимо всего прочего, автоматизировал операцию сложения чисел, имеющих более восьми разрядов. Поначалу этот перенос кажется просто небольшим трюком, использующимся при сложении, однако в случае с сумматорами перенос бита — основная проблема. Если вы разработали сумматор, который делает все, кроме переноса бита, считайте, что вы ни на шаг не приблизились к цели!

Ключевой фактор при оценке старых вычислительных машин — насколько успешно они справлялись с переносом. Например, конструкция механизма переноса, разработанная Паскалем, не позволяла машине производить вычитание. Для того чтобы вычесть одно число из другого, необходимо было дополнение до девяти, которое я продемонстрировал в главе 13. Эффективные механические калькуляторы, которыми могли пользоваться люди, появились только к концу XIX века.

Одним из любопытных изобретений, которому предстояло значительно повлиять на историю вычислений, а также на текстильную промышленность, был автоматический ткацкий станок, разработанный Жозефом Жаккаром (1752–1834). В станке Жаккара, созданном в 1804 году, для задания узора ткани использовались металлические карты с пробитыми в них отверстиями (вроде перфокарт для самоиграющих пианино). Величайшим достижением Жаккара

стал его черно-белый автопортрет в шелке, на создание которого потребовалось около десяти тысяч карт.

В XVIII веке *вычислителем* назывался человек, нанятый специально для того, чтобы производить вычисления. Большим спросом пользовались таблицы логарифмов, а таблицы тригонометрических функций широко применялись для астрономической навигации. Если вам нужно было опубликовать новый набор таблиц, приходилось нанимать многочисленных «компьютеров», организовывать их работу, а затем систематизировать полученные данные. Разумеется, ошибки могли возникнуть на любой стадии, начиная от расчетов и заканчивая подготовкой к печати.

Стремлением избавить математические таблицы от ошибок руководствовался в своей работе Чарльз Бэббидж (1791–1871), британский математик и экономист, практически современник Сэмюэла Морзе.

В то время математические таблицы (например, логарифмов) создавались не путем вычисления фактического логарифма для каждой записи. Это заняло бы слишком много времени. Вместо этого логарифмы вычислялись для избранных чисел, а логарифмы остальных чисел можно было найти довольно просто — путем интерполяции с использованием так называемых *разностей*.

В начале 1820-х годов Бэббидж считал, что он в состоянии разработать и сконструировать машину для автоматизации процесса составления таблицы, вплоть до подготовки к печати. Это устранило бы ошибки. Он придумал «разностную машину», которая по сути была большим механическим сумматором. Многоразрядные десятичные числа представлялись с помощью зубчатых колес, каждое из которых могло находиться в любом из десяти положений. Отрицательные числа обрабатывались с использованием дополнений до десяти. Несмотря на то что ранние модели доказали работоспособность конструкции Бэббиджа, разностная машина так никогда и не была завершена, поскольку создавалась на гранты британского правительства, которых, разумеется, никогда не хватало. В 1833 году Бэббидж прекратил работу над ней.

Однако к этому времени у Бэббиджа возникла лучшая идея. Она заключалась в создании так называемой аналитической машины, разработка конструкции которой занимала ученого до самой смерти, причем ему удалось фактически собрать несколько небольших моделей и частей этого механизма*.

---

* В результате британское правительство, щедро выдававшее Бэббиджу гранты и выстроившее для работ и размещения машины целое здание, так и не получило ни разностную машину, ни аналитическую в тридцатых годах XIX века… не получило оно ни одной машины и в пятидесятых. В июне 1952 года министр финансов Бенджамин Дизраэли принял финальное и безапелляционное решение о прекращении, а точнее, об отказе в возобновлении финансирования работ Бэббиджа. *Прим. науч. ред.*

Аналитическая машина — устройство, наиболее приближенное к компьютеру из всего, что было создано в XIX веке. Конструкция Бэббиджа предусматривала *хранилище* (концептуально напоминающее память компьютера) и *мельницу* (устройство для выполнения арифметических операций). Умножение можно было производить путем многократного сложения, а деление — путем многократного вычитания.

Самое интересное заключается в том, что аналитическую машину можно было запрограммировать с помощью карт наподобие тех, что использовались в ткацком станке Жаккара. Как выразилась Августа Ада Байрон, графиня Лавлейс (1815–1852), в примечаниях к своему переводу статьи, написанной итальянским математиком об аналитической машине Бэббиджа: «Можно сказать, что аналитическая машина плетет алгебраические узоры подобно тому, как ткацкий станок Жаккара плетет цветы и листья».

Бэббидж, кажется, был первым, кто осознал важность условных переходов в компьютерах. Снова приведем слова Ады Байрон: «Таким образом, *цикл* операций следует понимать как *любой набор операций*, выполняемый *более одного раза*. *Цикл* является циклом вне зависимости от того, повторяется он *дважды* или неопределенное число раз, поскольку именно *факт повторения* делает цикл тем, чем он является. Во многих анализируемых случаях существуют *рекуррентные группы*, состоящие из одного или нескольких циклов, то есть *цикл цикла* или *цикл циклов*».

Несмотря на то что разностная машина в конечном итоге была сконструирована Георгом Шютцем и его сыном Эдвардом в 1853 году, машины Бэббиджа оставались забытыми, о них вспомнили только в 1930-х, когда люди начали исследовать основы информатики. К тому времени все, чего достиг Бэббидж, уже было превзойдено более поздними технологиями, и он мало что мог предложить компьютерному инженеру XX века, кроме значительно опережающего свое время предвидения автоматизации.

Еще одним толчком для развития информатики послужила Конституция Соединенных Штатов Америки. Помимо всего прочего, в ней содержится призыв к проведению переписи населения каждые десять лет. При проведении переписи 1880 года собиралась информация о возрасте, поле и национальности. На анализ данных ушло около семи лет.

Опасаясь того, что анализ переписи 1890 года займет больше десятилетия, Бюро переписи населения изучило возможность ее автоматизации и выбрало механизм, придуманный Германом Холлеритом (1860–1929), который работал в качестве статистика в 1880 году.

Холлерит планировал использовать картонные перфокарты размером 168,278 × 82,551 мм. Маловероятно, что Холлерит знал о том, как Чарльз Бэббидж

использовал карты для программирования своей аналитической машины, однако он почти наверняка был знаком с использованием карточек в ткацком станке Жаккара. Отверстия в этих карточках были организованы в 24 столбца по 12 позиций, что в общей сложности давало 288 позиций. Эти позиции соответствовали определенным характеристикам человека, участвующего в переписи. Переписчик указывал эти особенности, пробивая прямоугольные отверстия размером в четверть дюйма в соответствующем месте карты.

Читая книгу, вероятно, вы настолько привыкли мыслить в терминах двоичных кодов, что могли предположить, что карта с 288 возможными отверстиями способна хранить 288 бит информации. Однако эти карты использовались не так.

Например, перфокарта, применяемая при переписи в чисто двоичной системе, имела бы одну позицию для пола. Она была бы либо пробита — в случае, если опрашиваемый — мужчина, либо не пробита — в случае, если это женщина (или наоборот). Однако карты Холлерита предусматривали две позиции для пола: одна пробивалась для мужчин, другая — для женщин. Аналогичным образом переписчик указывал возраст субъекта, пробивая два отверстия. Первое обозначало пятилетний диапазон: от 0 до 4, от 5 до 9, от 10 до 14 и т. д. Второе отверстие пробивалось в одной из пяти позиций для обозначения точного возраста в этом диапазоне. Для кодирования возраста требовались в общей сложности 28 позиций на карте. При использовании двоичной системы нужны были бы всего семь позиций для кодирования любого возраста от 0 до 127 лет.

Мы должны простить Холлерита за то, что он не внедрил двоичную систему для записи информации, собранной при переписи населения. Преобразование возраста в двоичные числа было непосильной задачей для тех, кто проводил перепись 1890 года. Кроме того, существует практическая причина, по которой использование перфокарт не может быть полностью основанным на двоичной системе. Двоичная система предполагает вероятность того, что будут пробиты все (или почти все) отверстия, что сделает карту чрезвычайно хрупкой.

Данные переписи собираются так, чтобы их можно было подсчитать, то есть обобщают в *таблицы*. Разумеется, вы хотите знать, сколько людей живет в том или ином районе, однако также интересно получить сведения о распределении населения по возрасту. Для этого Холлерит сконструировал табулятор — машину, в которой ручное управление сочеталось с автоматизацией. Оператор прижимал к каждой перфокарте пресс с 288 подпружиненными штырями. В тех местах карточки, где были пробиты отверстия, эти штыри погружались в резервуар с ртутью, что приводило к замыканию электрической цепи,

активировавшей электромагнит, который затем увеличивал на единицу значение десятичного счетчика.

Холлерит использовал электромагниты и в машине для сортировки перфокарт. Например, вам может понадобиться собрать отдельную возрастную статистику по каждой профессии. Сначала нужно сортировать карты по профессиям, затем отдельно для каждой из них собрать данные по возрастам. Сортировочная машина использовала тот же ручной пресс, что и табулятор, однако сортировщик применял электромагниты для того, чтобы открывать задвижки одного из 26 отделений. В это отделение оператор опускал карту и вручную закрывал задвижку.

Этот эксперимент по автоматизации переписи 1890 года оказался чрезвычайно успешным. В общей сложности было обработано более 62 миллионов карточек. Они содержали в два раза больше данных по сравнению с тем, что удалось собрать в ходе переписи 1880 года, а обработаны эти сведения были примерно в три раза быстрее. Холлерит и его изобретения стали известны во всем мире. В 1895 году он даже отправился в Москву и успешно продал свое оборудование для первой российской переписи 1897 года.

Герман Холлерит положил начало длинной последовательности событий. В 1896 году он основал компанию Tabulating Machine Company, занимающуюся сдачей в аренду и продажей оборудования для работы с перфокартами. К 1911 году в результате пары слияний она превратилась в Computing-Tabulating-Recording Company, или C-T-R. В 1915 году ее президентом стал Томас Джон Уотсон (1874–1956), который в 1924 году поменял название на International Business Machines Corporation, или IBM.

К 1928 году оригинальные карты, использовавшиеся в переписи 1890 года, превратились в знаменитые перфокарты IBM с 80 столбцами и 12 строками. Они продолжали активно использоваться на протяжении более 50 лет, и даже в последующие годы их иногда называли *картами Холлерита*. Об эволюции этих карт расскажу подробнее в главах 20, 21 и 24.

Прежде чем перенестись в двадцатое столетие, давайте убедимся, что у нас сложилось правильное представление об этой эпохе. По очевидным причинам в данной книге я уделял пристальное внимание изобретениям, которые являются цифровыми по своей природе. К ним относятся телеграф, азбука Брайля, машины Бэббиджа и карты Холлерита. При работе с цифровыми концепциями и устройствами вы легко можете подумать, что цифровым является весь мир. Однако открытия и изобретения XIX века были явно *не* цифровыми. Действительно, очень малая часть природного мира, который мы воспринимаем с помощью органов чувств, цифровая. Скорее, мир — это континуум, который нелегко представить с помощью чисел.

Несмотря на то что Холлерит использовал реле в своих карточных табуляторах и сортировщиках, компьютеры, созданные на основе реле, которые впоследствии стали называться *электромеханическими*, появились только в середине 1930-х годов. В этих машинах обычно использовались не телеграфные реле, а реле, разработанные для маршрутизации телефонных вызовов.

Эти первые релейные компьютеры не были похожи на то, что мы собирали в предыдущей главе (их конструкция основана на микропроцессорах, созданных в 1970-х). Сегодня для нас очевидно, что компьютеры должны использовать двоичные числа, однако так было не всегда.

Другое отличие нашего релейного компьютера от первых настоящих машин в том, что никто в 1930-х годах не был настолько сумасшедшим, чтобы собрать из реле память объемом 524 288 бит! Стоимость и требования к пространству и мощности делали невозможным создание такой памяти. Скудный объем доступной памяти использовался исключительно для хранения промежуточных результатов. Сами программы находились на физическом носителе, например на бумажной ленте с перфорацией. Действительно, наш процесс ввода кода и данных в память — более современная концепция.

Хронологически первый релейный компьютер, по-видимому, сконструировал Конрад Цузе (1910–1995), который в 1935 году, будучи студентом-инженером, начал собирать машину в квартире своих родителей в Берлине. Эта машина использовала двоичные числа, но в ее ранних версиях применялась механическая память, а не реле. Для программирования своих компьютеров Цузе пробивал отверстия в старой 35-миллиметровой кинопленке.

В 1937 году Джордж Стибиц (1904–1995) из Bell Telephone Laboratories принес домой пару телефонных реле и собрал на своем кухонном столе однобитный сумматор, который его жена позднее назвала «К-машиной» («К» — значит «кухня»). Этот эксперимент лег в основу компьютера Complex Number Computer, созданного в Bell Labs в 1939 году.

Между тем студент выпускного курса Гарварда Эйкен (1900–1973) искал способ выполнения множества однообразных вычислений, что привело к сотрудничеству Гарварда и IBM, в результате которого был создан автоматический вычислитель, управляемый последовательностями, впоследствии получивший имя «Марк I». Работа над этим устройством была завершена в 1943 году. Этот первый цифровой компьютер, способный печатать таблицы, наконец реализовал мечту Чарльза Бэббиджа. Компьютер «Марк II» был самой крупной релейной машиной, использующей 13 тысяч реле. В Гарвардской вычислительной лаборатории, возглавляемой Эйкеном, впервые был прочитан курс информатики.

Реле подходили для создания компьютеров, но были неидеальны. Поскольку они были механическими, их работа основывалась на изгибании металлической

пластины. После продолжительной работы реле могли сломаться, а также выйти из строя из-за частичек грязи или бумаги, застрявших между контактами. Известен случай, когда в 1947 году из реле компьютера «Марк II» в Гарварде была извлечена мошка. Грейс Хоппер (1906–1992), сотрудничавшая с Эйкеном с 1944 года, а позднее ставшая известным специалистом в области языков программирования, приклеила эту мошку в журнал с пометкой: «Первый отловленный баг».

Возможная замена для реле — вакуумная лампа, разработанная Джоном Флемингом (1849–1945) и Ли де Форестом (1873–1961) для радио. К началу 1940-х годов вакуумные лампы повсеместно использовались для усиления телефонных сигналов. Практически в каждом доме был радиоприемник, наполненный светящимися трубками, которые усиливали радиосигналы, обеспечивая их слышимость. Как и в случае с реле, из вакуумных ламп можно собрать вентили И, ИЛИ, И-НЕ и ИЛИ-НЕ.

Не важно, из чего собраны вентили — из реле или из вакуумных ламп. Вентили всегда можно объединить в сумматоры, селекторы, дешифраторы, триггеры и счетчики. Все, что я говорил о компонентах на основе реле в предшествующих главах, остается в силе при замене реле вакуумными лампами.

Тем не менее у вакуумных ламп наблюдались свои недостатки: они были дорогими, потребляли много электричества и выделяли много тепла. Самая большая проблема заключалась в том, что лампы в итоге перегорали. Это был факт, с которым людям приходилось мириться. Владельцы ламповых радиоприемников привыкли к необходимости периодически заменять в них лампы. Телефонная система была спроектирована с избыточной надежностью, поэтому периодически перегорающие лампы не представляли большой проблемы (в любом случае никто не ожидает от телефонной системы безупречной работы). А вот если лампа перегорает в компьютере, это можно обнаружить не сразу. Кроме того, в компьютере используется так *много* вакуумных ламп, что они могут перегорать в среднем каждые несколько минут.

Большое преимущество использования вакуумных ламп по сравнению с реле в том, что лампы могут переключаться из одного состояния в другое примерно за одну миллионную долю секунды — за одну *микросекунду*. Вакуумная лампа изменяет состояние (включается или выключается) в тысячу раз быстрее, чем реле, которое переключается из одного состоянии в другое в лучшем случае примерно за одну миллисекунду, то есть тысячную долю секунды. Интересно отметить, что скорость не представляла серьезной проблемы на ранних этапах развития компьютерной индустрии, поскольку общая скорость вычислений была связана со скоростью, с которой машина считывала программу с бумажной или пластиковой ленты. Пока компьютеры

работали на этом принципе, преимущество вакуумных ламп по сравнению с реле не имело значения.

Начиная с 1940-х годов вакуумные лампы стали вытеснять реле в конструкции новых компьютеров. К 1945 году реле совсем перестали использоваться. В то время как релейные машины назывались электромеханическими компьютерами, вакуумные лампы стали основой для первых *электронных* компьютеров.

В 1943 году в Великобритании начал работать компьютер «Колосс», использовавшийся для расшифровки сообщений, созданных с помощью немецкой шифровальной машины «Энигма». Над этим проектом (и некоторыми более поздними британскими компьютерами) среди прочих работал Алан Тьюринг (1912–1954), который в наши дни известен двумя статьями. В первой, опубликованной в 1937 году, он ввел понятие «вычислимость» — анализ того, что могут и чего не могут сделать компьютеры. Он также разработал абстрактную модель компьютера, которая теперь известна под названием машины Тьюринга. Вторая известная статья была посвящена искусственному интеллекту. Автор представил тест для машинного интеллекта — тест Тьюринга.

В Электротехнической школе Мура при Пенсильванском университете Джон Эккерт (1919–1995) и Джон Моучли (1907–1980) разработали компьютер ENIAC (Electronic Numerical Integrator and Computer, электронный числовой интегратор и вычислитель). В нем использовались 18 тысяч вакуумных ламп, и компьютер был закончен в конце 1945 года. ENIAC, вес которого составлял около 30 тонн, можно считать самым большим компьютером в истории. К 1977 году в продаже уже были гораздо более быстрые компьютеры. Однако Эккерт и Моучли не смогли запатентовать машину из-за заявки их конкурента Джона Атанасова (1903–1995), собравшего электронный компьютер раньше, который, однако, так никогда и не заработал.

Компьютер ENIAC привлек внимание математика Джона фон Неймана (1903–1957). Родившийся в Венгрии, фон Нейман проживал в Соединенных Штатах с 1930 года. Выдающийся человек, известный своей способностью выполнять в уме сложнейшие арифметические операции, фон Нейман был профессором математики в Принстонском институте перспективных исследований и изучал все — от квантовой механики до применения теории игр в экономике.

Джон фон Нейман помог разработать компьютер EDVAC (Electronic Discrete Variable Automatic Computer, электронный автоматический вычислитель с дискретными переменными), являвшийся усовершенствованной версией компьютера ENIAC. В статье* 1946 года «Предварительное обсуждение логической

* Работы фон Неймана публиковались в том числе и в СССР, например см.: *Нейман Дж.* Вычислительная машина и мозг // Кибернетический сборник № 1: сборник переводов. М.: Издательство иностранной литературы, 1960. *Прим. науч. ред.*

конструкции электронной вычислительной машины», написанной в соавторстве с Артуром Берксом и Германом Голдстайном, он описал несколько особенностей компьютера, благодаря которым машина EDVAC значительно превосходила ENIAC. Разработчики EDVAC решили, что компьютер должен использовать двоичную систему счисления. В машине ENIAC использовалась десятичная. Кроме того, компьютер должен обладать максимально возможным объемом памяти, и эта память должна хранить и программный код, и данные, получаемые в процессе работы. С компьютером ENIAC дело обстояло не так. Программирование ENIAC осуществлялось с помощью переключателей и соединения кабелей. Эти инструкции должны были храниться в памяти последовательно и адресоваться с помощью счетчика команд, при этом допускались условные переходы. Такой принцип стал известен как *концепция запоминаемой программы*.

Эти принципы были таким важным шагом в развитии информатики, что сегодня мы говорим о них как об *архитектуре фон Неймана*. Компьютер, который мы собрали в предыдущей главе, представляет собой классическую машину фон Неймана. Однако у архитектуры фон Неймана есть узкое место. Машина фон Неймана обычно тратит значительное время на извлечение инструкций из памяти при подготовке к их выполнению. Напомним, что окончательная конструкция компьютера из главы 17 предполагала, что при работе с той или иной инструкцией три четверти времени затрачивается на ее извлечение.

Во времена компьютера EDVAC было нецелесообразно создавать из вакуумных ламп память большого объема. Вместо этого было предложено несколько весьма странных решений. Среди успешных было использование *памяти с ртутной линией задержки*, в которой применялись пятифутовые трубки с ртутью. С одного конца трубки с интервалом около одной микросекунды в ртуть посылались слабые импульсы. За одну миллисекунду они достигали другого конца трубки, где детектировались как звуковые волны и отправлялись обратно. Таким образом, каждая трубка с ртутью могла хранить около 1024 бит информации.

Только в середине 1950-х годов была разработана память, состоявшая из больших массивов маленьких намагниченных металлических колец, через которые проходили провода. Каждое такое кольцо могло хранить один бит информации.

Джон фон Нейман был не единственным человеком, который размышлял о природе компьютеров в 1940-х годах.

Клод Шеннон также был влиятельным мыслителем. В главе 11 я обсуждал его магистерскую диссертацию 1938 года, в которой была установлена взаимосвязь между переключателями, реле и булевой алгеброй. В 1948 году, работая в Bell Telephone Laboratories, он опубликовал в Bell System Technical Journal статью «Математическая теория связи», где не только впервые употребил в печати слово «бит», но и заложил основы раздела науки, известной сегодня как *теория информации*.

Теория информации изучает возможность передачи цифровой информации при наличии шума (который обычно препятствует передаче всей информации), а также способы его компенсации. В 1949 году Шеннон написал первую статью о программировании компьютера для игры в шахматы, а в 1952 году разработал механическую мышь, управляемую реле, которая могла находить выход из лабиринта. Помимо всего прочего, в Bell Labs Шеннон был хорошо известен еще и своим умением ездить на одноколесном велосипеде, при этом жонглируя.

Норберт Винер (1894–1964), который в возрасте 18 лет в Гарварде получил степень доктора философии по математике, наиболее известен благодаря своей книге «Кибернетика, или Управление и связь в животном и машине» (1948). Винер придумал название «*кибернетика*» (от греческого «кормчий») для теории о взаимосвязи биологических процессов в людях и животных с механикой компьютеров и роботов. В поп-культуре вездесущая приставка «*кибер-*» теперь обозначает все, что имеет отношение к компьютерам. В частности, миллионы связанных через интернет компьютеров называются *киберпространством*. Это слово было придумано писателем Уильямом Гибсоном и появилось в романе 1984 года «Нейромант», написанном в жанре *киберпанка*.

В 1948 году компания Eckert-Mauchly Computer Corporation (позднее вошедшая в Remington Rand) начала работу над тем, чему предстояло стать первым доступным широкой аудитории компьютером, — UNIVAC (Universal Automatic Computer, универсальный автоматический компьютер). Он был закончен в 1951 году, а первый экземпляр был доставлен в Бюро переписи населения. Компьютер UNIVAC дебютировал в эфире канала CBS, где использовался для прогнозирования результатов президентских выборов 1952 года. Уолтер Кронкайт называл его электронным мозгом. В 1952 году компания IBM объявила о выпуске первой коммерческой компьютерной системы — 701.

Так началась долгая история использования компьютеров корпорациями и правительством. Какой бы интересной ни была эта история, сейчас мы перейдем к обсуждению другой тенденции, зародившейся в 1947 году, которая позволила уменьшить стоимость и размер компьютеров, превратив их в бытовую технику. Этот прорыв в области электроники едва не остался незамеченным.

Корпорация Bell Telephone Laboratories на протяжении многих лет была местом, где умные люди имели возможность работать практически над любым интересовавшим их проектом. К счастью, некоторые из них увлекались компьютерами. Я уже упоминал Джорджа Стибица и Клода Шеннона, которые внесли значительный вклад в развитие вычислительной техники, работая в Bell Labs. Позднее, в 1970-х годах, в Bell Labs была разработана компьютерная операционная система Unix и язык программирования C, о которых расскажу позднее.

Корпорация Bell Labs возникла 1 января 1925 года, когда компания American Telephone and Telegraph официально отделила свои научные и технические исследовательские подразделения от остальной части бизнеса, создав дочернее предприятие. Основная цель Bell Labs заключалась в разработке технологий для улучшения работы телефонной системы. К счастью, это поручение было достаточно туманным и предполагало всевозможные направления исследований, в том числе очевидную и не теряющую своей актуальности задачу, связанную с усилением передаваемого по проводам звукового сигнала без его искажения.

Начиная с 1912 года компания Bell System работала над ламповыми усилителями. Значительная часть исследований и разработок была направлена на усовершенствование вакуумных ламп с целью их использования в телефонной системе. Несмотря на проделанную работу, вакуумные лампы по-прежнему оставляли желать лучшего. Они были большими, потребляли много электроэнергии и со временем перегорали. Однако у них не было альтернативы.

Все изменилось 16 декабря 1947 года, когда два физика из Bell Labs, Джон Бардин (1908–1991) и Уолтер Браттейн (1902–1987), собрали усилитель другого типа из германиевой пластины — элемента, известного как *полупроводник*, — и полоски золотой фольги. Через неделю они продемонстрировали усилитель своему шефу Уильяму Шокли (1910–1989). Это был первый *транзистор*, устройство, которое некоторые считают самым важным изобретением XX века.

Транзистор появился не на пустом месте. За восемь лет до этого, 29 декабря 1939 года, Шокли написал в своей записной книжке: «Сегодня мне пришло в голову, что в принципе можно создать усилитель, использующий вместо вакуумных ламп полупроводники». После демонстрации первого транзистора много лет ушло на его доработку. Только в 1956 году Шокли, Бардин и Браттейн получили Нобелевскую премию по физике «за исследования полупроводников и открытие транзисторного эффекта».

Как вы помните, электроны в атоме распределены по оболочкам, окружающим ядро. Медь, золото и серебро характеризуются наличием только одного электрона на внешней оболочке их атомов. Этот электрон легко может оторваться от остальной части атома и двигаться, создавая электрический ток. Противоположностью проводников являются изоляторы, например резина и пластик, которые практически не проводят электричество.

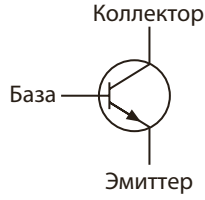
Германий и кремний (а также некоторые соединения) называются *полупроводниками* не потому, что они проводят электричество в два раза хуже, чем проводники, а потому, что их проводимостью можно управлять различными способами. Полупроводники имеют четыре электрона на внешней оболочке атома, что составляет половину от их максимально возможного количества. В чистом полупроводнике атомы образуют очень устойчивые связи, создавая

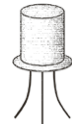
кристаллическую решетку, подобную кристаллической решетке алмаза. Такие полупроводники проводят электричество не очень хорошо.

Однако полупроводники можно *легировать*, то есть добавить в них некоторые примеси. Один тип примесей добавляет дополнительные электроны к тем, которые необходимы для создания связи между атомами. Они называются *полупроводниками n-типа* (n — от английского negative — «отрицательный»). В результате добавления другого типа примесей получается *полупроводник p-типа* (p — от positive — «положительный»).

Чтобы создать усилитель из полупроводников, нужно между двумя слоями полупроводника n-типа расположить прослойку из полупроводника p-типа. Получившееся устройство называется NPN-транзистором, а тремя его составными частями являются *коллектор*, *база* и *эмиттер*.

На рисунке приведено схематическое изображение NPN-транзистора.





Небольшое напряжение на базе управляет гораздо большим током, проходящим от коллектора к эмиттеру. При отсутствии напряжения на базе транзистор закрывается.

Как правило, транзисторы имеют вид небольших металлических цилиндров диаметром примерно 6,4 миллиметра с тремя проводами.

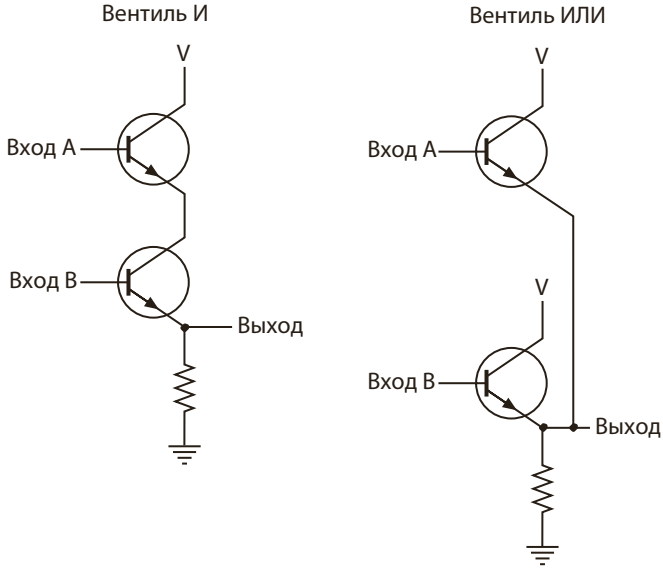
Разработка транзистора ознаменовала начало эры *твердотельной* электроники, которая называется так потому, что транзисторы не требуют использования вакуумных ламп и создаются из твердых веществ, в частности из полупроводников, чаще всего из кремния (в наши дни). Помимо того, что по сравнению с вакуумными лампами транзисторы имеют куда меньшие размеры, они потребляют гораздо меньше электроэнергии, генерируют меньше тепла и дольше служат. Носить в кармане ламповый радиоприемник было невозможно. Транзисторный радиоприемник может питаться от небольшой батарейки и, в отличие от лампового, не нагревается. Ношение транзисторного радиоприемника в кармане стало возможным для некоторых счастливчиков, распаковавших подарки в рождественское утро 1954 года. В этих первых карманных радиоприемниках использовались транзисторы, выпущенные компанией Texas Instruments, сыгравшей важную роль в полупроводниковой революции.

Однако *первым* коммерческим применением транзисторов было их использование в слуховых аппаратах. В память о многолетней работе Александра Белла с глухими людьми корпорация AT&T разрешила изготовителям слуховых аппаратов применять транзисторные технологии, не платя за использование патентов. Первый транзисторный телевизор был выпущен в 1960 году, и сегодня ламповых приборов практически не найти. (Хотя некоторые меломаны и электрогитаристы по-прежнему предпочитают ламповые усилители их транзисторным аналогам.)

В 1956 году Шокли покинул Bell Labs, чтобы основать компанию Shockley Semiconductor Laboratories. Он переехал в Пало-Альто (Калифорния), где вырос. Его компания стала первым местным предприятием, работающим в данном направлении. Со временем в этой местности появились другие полупроводниковые и компьютерные компании, а область к югу от Сан-Франциско теперь неофициально называется Кремниевой долиной.

Вакуумные лампы изначально разрабатывались для применения в усилителях, однако их также можно было использовать в качестве переключателей в логических вентилях. То же самое касается транзистора. На следующем рисунке изображен вентиль И на основе транзисторов, структура которого напоминает версию с реле. Только когда входы A и B равны логической единице, то есть на базу подается положительное напряжение, оба транзистора проводят ток, а выход равен 1. Резистор при этом предотвращает короткое замыкание.

Соединив два транзистора так, как показано на схеме справа, вы получите вентиль ИЛИ. В вентиле И эмиттер верхнего транзистора соединен с коллектором нижнего. В вентиле ИЛИ коллекторы обоих транзисторов подключены к источнику питания. Эмиттеры соединены между собой.



Как видите, все, что мы узнали о создании логических вентилей и других компонентов из реле, справедливо и для транзисторов. Реле, лампы и транзисторы изначально разрабатывались в основном для усилителей, однако из них можно собрать логические вентили для компьютеров. Первые транзисторные компьютеры были созданы в 1956 году, и спустя несколько лет лампы перестали применяться.

Транзисторы, безусловно, делают компьютеры более надежными, компактными и экономичными. Но облегчают ли они процесс *сборки*?

На самом деле нет. Разумеется, транзистор позволяет уместить больше логических вентилей в меньшем пространстве, однако вам по-прежнему придется беспокоиться о *соединении* всех этих компонентов. Соединить транзисторы в логические вентили так же сложно, как реле и вакуумные лампы. Этот процесс усложняется еще и меньшим размером, а также тем, что транзисторы труднее держать. Если бы вы решили собрать компьютер, описанный в главе 17, и массив RAM емкостью 64 килобайт из транзисторов, то бóльшая часть времени на этапе проектирования была бы потрачена на разработку некой структуры, где бы крепились все компоненты. Основной физический труд сводился бы к утомительному соединению миллионов транзисторов.

Как мы уже выяснили, существуют определенные комбинации часто встречающихся транзисторов. Пары транзисторов почти всегда соединены в вентили. Из вентилей часто собираются триггеры, сумматоры, селекторы или дешифраторы. Триггеры объединяются в многобитные защелки или массивы RAM. Собрать компьютер было бы проще, если бы транзисторы были предварительно объединены в распространенные конфигурации.

Эту идею, по-видимому, впервые предложил британский физик Джеффри Даммер, который в ходе выступления в мае 1952 года сказал: «Я хотел бы заглянуть в будущее. С появлением транзистора и работ по полупроводникам в целом сегодня, по-видимому, можно ставить вопрос о создании электронного оборудования в виде твердого блока без каких-либо соединительных проводов. Этот блок может состоять из слоев изолирующих, проводящих, преобразующих сигнал из переменного в постоянный и усиливающих сигнал материалов. Задание электронных функций компонентов и их соединение должным образом может быть выполнено путем вырезания участков отдельных слоев».

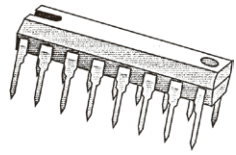
Однако на создание работающего продукта ушло еще несколько лет.

Ничего не зная о прогнозе Даммера, в июле 1958 года Джек Килби из компании Texas Instruments подумал, что на одном кристалле кремния можно объединить несколько транзисторов, а также резисторы и другие электрические компоненты. Шесть месяцев спустя, в январе 1959 года, практически та же идея возникла у Роберта Нойса (1927–1990). Сначала Нойс работал в компании

Shockley Semiconductor Laboratories, но в 1957 году он и еще семеро ученых покинули ее, чтобы основать корпорацию Fairchild Semiconductor Corporation.

В сфере технологий одновременное изобретение — довольно распространенное явление. Несмотря на то что Килби изобрел свое устройство за шесть месяцев до Нойса, а компания Texas Instruments подала заявку на патент раньше, чем Fairchild Semiconductor, Нойс получил патент первым. Последовавшие за этим судебные тяжбы завершились с устраивающим всех результатом только спустя десять лет. Несмотря на то что Килби и Нойс никогда не работали вместе, сегодня они считаются соавторами *интегральной микросхемы* (ИС), обычно называемой *чипом*.

Создание интегральных схем — сложный процесс, который предполагает наслаивание тонких пленок легированного кремния, протравленных в разных местах для образования микроскопических компонентов. Несмотря на то что разработка новой интегральной микросхемы предполагает большие затраты, массовое производство позволяет снизить цены: чем больше производится микросхем, тем дешевле они становятся.



Кремниевый чип очень тонкий и хрупкий, поэтому он должен быть надежно защищен корпусом, позволяющим в то же время соединить его компоненты с другими чипами. Чаще всего интегральные микросхемы помещаются в прямоугольный пластиковый корпус *DIP* (dual inline package, корпус с двухрядным расположением штырьковых выводов) с 14, 16 или даже 40 выводами.

Вот чип с 16 выводами. Если вы возьмете его так, чтобы небольшая выемка находилась слева (как показано на рисунке), то выводы будут нумероваться с 1 по 16 против часовой стрелки, начиная с вывода в левом нижнем углу и заканчивая выводом в левом верхнем углу. Штырьки расположены на расстоянии 2,5 миллиметра друг от друга.

На протяжении 1960-х годов рынок интегральных микросхем развивался благодаря космической программе и гонке вооружений. Первым массовым коммерческим продуктом, включавшим интегральную микросхему, был слуховой аппарат, распространявшийся компанией Zenith в 1964 году. В 1971 году компания Texas Instruments начала продавать первый карманный калькулятор, а компания Pulsar — первые цифровые часы. (Очевидно, в цифровых часах корпус ИС отличается от того, что мы обсуждали в вышеприведенном примере.) Вслед за ними появилось множество других товаров, в конструкцию которых входили интегральные микросхемы.

В 1965 году Гордон Мур (в то время сотрудник компании Fairchild Semiconductor, а позднее соучредитель корпорации Intel) заметил, что технология развивается так, что начиная с 1959 года количество транзисторов, которые

могут уместиться в одной микросхеме, ежегодно удваивается, и предсказал сохранение этой тенденции. Фактически такая технология развивалась немного медленнее, поэтому закон Мура (как он стал в конечном счете называться) был скорректирован и прогнозировал удвоение количества транзисторов в микросхеме каждые 18 месяцев. Это по-прежнему удивительно быстрое развитие, и закон Мура объясняет, почему домашние компьютеры устаревают всего за несколько лет.

На начальных этапах развития технологии о микросхемах, включающих в себя менее десяти логических вентилей, говорили как о схемах с *малым уровнем интеграции*. Схемы со *средним уровнем интеграции* (средние интегральные схемы, СИС) включали в себя от 10 до 100 вентилей, а схемы с *высоким уровнем интеграции* (большие интегральные схемы, БИС) — от 100 до 5000 вентилей. Затем были введены такие понятия, как *сверхвысокий уровень интеграции* (сверхбольшая интегральная схема, СБИС) — от 5 до 50 тысяч вентилей, *суперсверхвысокий уровень интеграции* — от 50 до 100 тысяч вентилей и *ультравысокий уровень интеграции* — более 100 тысяч вентилей.

Оставшуюся часть этой главы и всю следующую я предлагаю провести в середине 1970-х, в той древней эпохе, когда никто еще не слышал о фильме «Звездные войны», а схемы СБИС еще только маячили на горизонте. В то время для изготовления компонентов интегральных схем использовалось несколько различных технологий, каждая из которых определяет *семейство* ИС. К середине 1970-х годов преобладали семейства ТТЛ и КМОП.

Аббревиатура ТТЛ расшифровывается как *транзисторно-транзисторная логика*. Если бы в середине 1970-х вы работали инженером-разработчиком цифровых ИС (собирали из ИС более крупные схемы), то вашей настольной книгой был бы справочник по ТТЛ-микросхемам The TTL Data Book for Design Engineers, впервые опубликованный в 1973 году компанией Texas Instruments. Он содержал подробное описание интегральных микросхем ТТЛ серии 7400, продаваемых Texas Instruments и некоторыми другими компаниями, называемых так потому, что номер каждой ИС в этом семействе начинался с 74.

Каждая интегральная схема серии 7400 состоит из логических вентилей, сконфигурированных определенным образом. Некоторые микросхемы — простые логические вентили, из которых можно создать более крупные компоненты; другие — готовые компоненты: триггеры, сумматоры, селекторы и дешифраторы.

Первая ИС серии 7400, имеющая номер 7400, описана в справочнике The TTL Data Book как «счетверенная двухвходовая положительная схема И-НЕ». Это означает, что данная конкретная интегральная схема имеет четыре двухвходовых вентиля И-НЕ. Вентили И-НЕ называются *положительными*, поскольку наличие напряжения соответствует значению 1, а его отсутствие — значению 0.

На следующем рисунке изображена микросхема с 14 выводами и показано, как эти выводы соотносятся со входами и выходами.

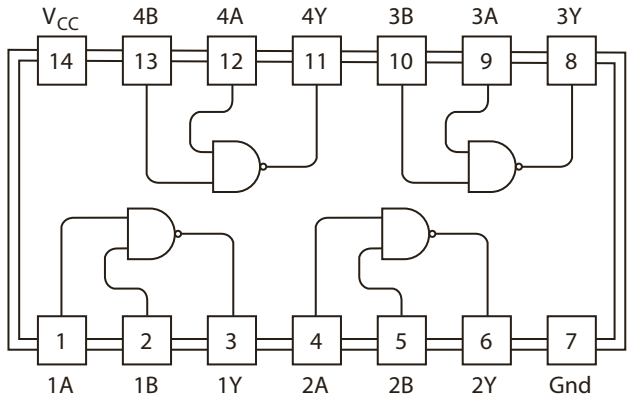


Диаграмма — это вид микросхемы сверху (выводы направлены вниз), при этом выемка в корпусе (упомянутая чуть ранее) расположена слева.

Вывод 14 обозначен символами V_{CC} и эквивалентен символу V, который я использовал для обозначения напряжения. По традиции любой двойной подстрочный буквенный индекс рядом с буквой V — источник питания. Буква *C* в этом индексе — это вход *коллектора* транзистора, на который подается напряжение. Вывод 7 обозначен буквами GND, что значит «земля» (ground). Каждая интегральная микросхема, которую вы используете, должна быть подключена к источнику питания и земле.

Для микросхем ТТЛ серии 7400 значение V_{CC} должно составлять от 4,75 до 5,25 вольта. Другими словами, питающее напряжение — это пять вольт ± 5%. Если напряжение упадет ниже 4,75 вольта, чип может перестать работать. Если оно превысит значение 5,25, чип может выйти из строя. Обычно для питания микросхем ТТЛ нельзя использовать батарейки. Даже если удастся найти пятивольтовую батарейку, недостаточная точность напряжения сделает ее неподходящим источником питания для этих чипов. Как правило, микросхемы ТТЛ требуют питания от розетки.

Каждый из четырех вентилей И-НЕ микросхемы 7400 имеет два входа и один выход. Они работают независимо друг от друга. В предыдущих главах мы говорили, что входной сигнал может иметь либо значение 1 (при наличии напряжения), либо значение 0 (при отсутствии напряжения). В действительности входной сигнал одного из этих вентилей И-НЕ может варьироваться от нуля вольт (земля) до пяти вольт (V_{CC}). В микросхеме ТТЛ напряжение, находящееся в диапазоне от 0 до 0,8 вольта, соответствует логическому нулю, а напряжение

от двух до пяти вольт — логической единице. Напряжения от 0,8 до 2,0 вольта следует избегать.

Напряжение на выходе вентиля ТТЛ, составляющее около 0,2 вольта, обычно соответствует логическому нулю, а 3,4 вольта — логической единице. Поскольку эти значения могут несколько отклоняться, то, говоря о входах и выходах интегральных схем, иногда вместо 0 и 1 люди используют такие понятия, как *низкий* и *высокий* уровень сигнала. Более того, низкое напряжение может означать логическую единицу, а высокое — логический ноль. Такая конфигурация характеризуется *отрицательной* логикой. В названии «счетверенная двухвходовая положительная схема И-НЕ» слово «положительная» означает схему с положительной логикой.

Значения напряжения на выходе вентиля ТТЛ 0,2 вольта (логический ноль) и 3,4 вольта (логическая единица) находятся в допустимых пределах — от 0 до 0,8 для логического нуля и от двух до пяти вольт для логической единицы. Таким образом микросхемы ТТЛ изолируются от *шумов*. Единичный выходной сигнал может уменьшиться примерно на 1,4 вольта, но по-прежнему останется достаточно высоким, чтобы его можно было квалифицировать в качестве входного единичного сигнала. Нулевой выходной сигнал может увеличиться на 0,6 вольта, но останется достаточно низким, чтобы категоризировать входной нулевой сигнал.

Вероятно, самым важным параметром конкретной интегральной схемы является *время установки*. Это время, необходимое для того, чтобы изменение входного сигнала привело к изменению выходного.

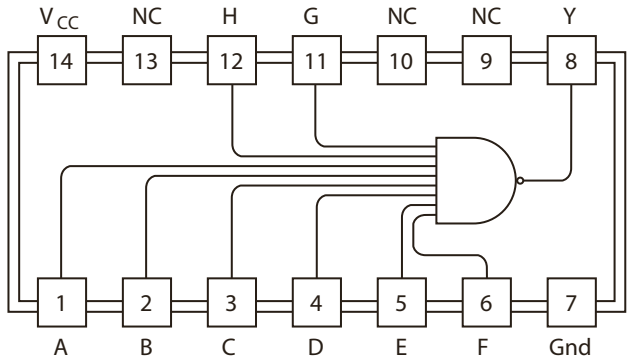
Время установки микросхем обычно измеряется в *наносекундах*. Наносекунда — очень короткий промежуток времени. Одна тысячная часть секунды — это миллисекунда. Миллионная часть секунды — микросекунда. Наносекунда — это одна миллиардная часть секунды. Время установки для вентилей И-НЕ в микросхеме 7400 гарантированно составляет менее 22 наносекунд. Это 0,000000022 секунды, или 22 миллиардные доли секунды.

Если вам сложно представить такой маленький промежуток времени, вы не одиноки. Мы можем охватить его лишь мыслью. Наносекунды намного короче всего, что доступно человеческому опыту, поэтому они навсегда останутся за пределами нашего понимания. Каждое объяснение лишь делает наносекунду более непостижимой. Например, я могу сказать, что если вы держите эту книгу на расстоянии 30 сантиметров от лица, то наносекунда — это время, за которое свет преодолевает расстояние от страницы до вашего глаза. Однако стали ли вы от этого лучше понимать, что такое наносекунда?

Тем не менее именно благодаря таким коротким промежуткам времени, как наносекунда, возможно существование компьютеров. Как вы видели в главе 17, компьютерный процессор выполняет очень простые действия: перемещает

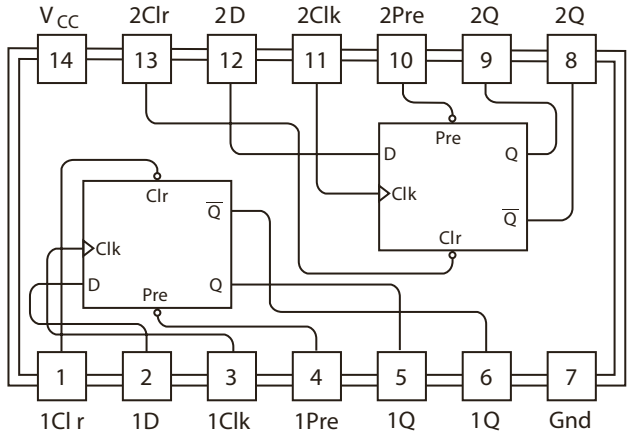
байт из памяти в регистр, складывает с другим байтом и возвращает результат обратно в память. Единственная причина, по которой результат работы компьютера — нечто существенное (в данном случае речь идет о реальном компьютере, а не о том, который описывался в главе 17), заключается в том, что эти операции происходят очень быстро. Как сказал Роберт Нойс: «Если примириться с понятием наносекунды, то компьютерные операции концептуально довольно просты».

Давайте продолжим изучение справочника по микросхемам ТТЛ. В этой книге вы увидите много уже знакомых компонентов. Микросхема 7402 содержит четыре двухвходовых вентиля ИЛИ-НЕ, микросхема 7404 — шесть инверторов, микросхема 7408 — четыре двухвходовых вентиля И, микросхема 7432 — четыре двухвходовых вентиля ИЛИ, а микросхема 7430 — восьмивходовый вентиль И-НЕ.

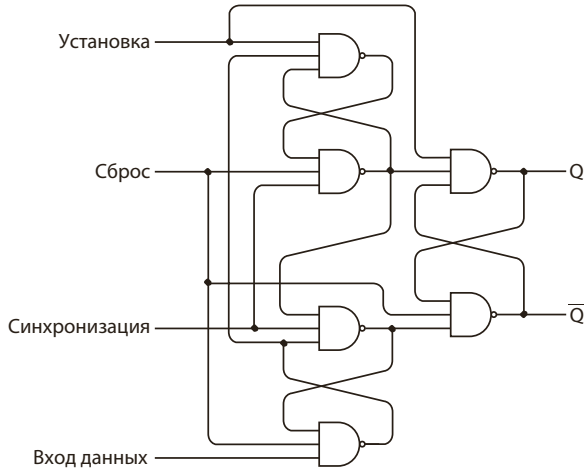


Аббревиатура *NC* означает no connection — «не подключено».

Микросхема 7474 тоже может показаться знакомой. Это сдвоенный D-триггер со сбросом и предустановкой, срабатывающий по фронту, схема которого выглядит следующим образом.



В справочник по микросхемам ТТЛ включена логическая схема для каждого из триггеров.



Эта схема может показаться похожей на схему, приведенную в конце главы 14, за исключением того, что я использовал вентили ИЛИ-НЕ. Приведенная в справочнике по микросхемам ТТЛ таблица логики также немного отличается.

Входы				**Выходы**	
Pre	**Clr**	**Clk**	**D**	**Q**	$\overline{\mathbf{Q}}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	L	↑	H	H	L
H	H	↑	L	L	H
H	H	L	H	Q_0	$\overline{Q}_0$

В этой таблице *H* означает *высокий* (high), а *L* — *низкий* (low) уровень сигнала. Если хотите, можете считать эти обозначения единицей и нулем. В моем триггере входы «Сброс» и «Установка» обычно равны 0; в данном случае они обычно равны 1.

Далее в справочнике по микросхемам ТТЛ вы обнаружите, что микросхема 7483 — это 4-битный двоичный полный сумматор, микросхема 74151 — селектор с восемью входами и одним выходом, 74154 — дешифратор с четырьмя входами и 16 выходами, 74161 — синхронный 4-разрядный двоичный счетчик, 74175 — счетверенный D-триггер со сбросом. Вы можете использовать две из перечисленных микросхем для создания 8-битной защелки.

Итак, теперь вы знаете, откуда взялись различные компоненты, которые я использовал в главе 11, — из справочника по микросхемам ТТЛ.

Будучи инженером-разработчиком цифровых ИС, вы потратили бы множество часов на чтение справочника по микросхемам ТТЛ и изучение существующих чипов. Освоив инструменты, вы могли бы собрать из микросхем ТТЛ компьютер, описанный в главе 17. Соединить между собой микросхемы намного проще, чем отдельные транзисторы. Однако вы вряд ли бы захотели использовать схемы ТТЛ для создания массива RAM объемом 64 килобайт. Объем самого емкого чипа RAM, описанного в справочнике The TTL Data Book for Design Engineers 1973 года, составлял всего 256 × 1 бит. Для создания массива RAM объемом 64 килобайт вам понадобилось бы 2048 таких чипов! Микросхемы ТТЛ никогда не были оптимальной технологией для создания памяти. К этой теме я вернусь в главе 21.

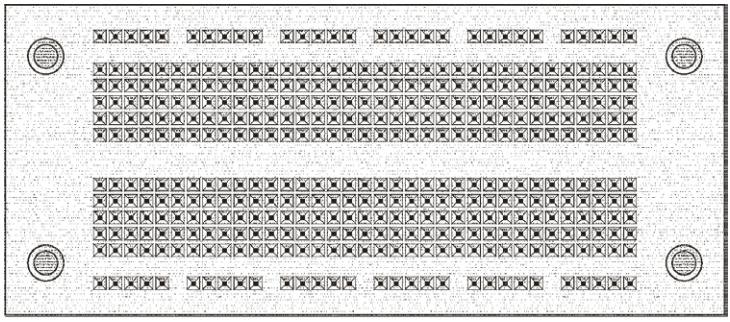
Вероятно, вы решите использовать осциллятор. Несмотря на возможность подключения выхода ТТЛ-инвертора к его же входу, лучше иметь осциллятор с более предсказуемой частотой. Такой осциллятор можно легко собрать, используя кристалл кварца, который помещается в небольшой плоский цилиндрический корпус с двумя выводами. Эти кристаллы вибрируют с определенной частотой, которая обычно составляет по меньшей мере миллион циклов в секунду. Миллион циклов в секунду соответствует частоте один *мегагерц*. Если бы компьютер, описанный в главе 17, был собран из микросхем ТТЛ, он бы нормально работал с тактовой частотой десять мегагерц. На выполнение каждой инструкции уходило бы 400 наносекунд. Это, безусловно, многократно превышает скорость работы релейных устройств.

Другим популярным семейством чипов является *КМОП* (комплементарная структура металл — оксид — полупроводник), или CMOS (complementary metal-oxide-semiconductor). Если бы в середине 1970-х в свое свободное время вы собирали схемы из чипов КМОП, то в качестве справочника могли бы использовать книгу CMOS Databook, опубликованную компанией National Semiconductor. Эта книга содержит информацию о микросхемах КМОП серии 4000.

Потребляемая мощность микросхем ТТЛ — от 4,75 до 5,25 вольта, для микросхем КМОП — от 3 до 18 вольт. Довольно большой диапазон! Кроме того, микросхемы КМОП потребляют гораздо меньше энергии по сравнению с ТТЛ-чипами, что делает возможным создание на их основе небольших устройств, работающих от батареек. Недостаток микросхемы КМОП — низкая скорость работы. Например, гарантированное время установки 4-битного полного сумматора КМОП 4008, работающего от напряжения 5 вольт, — 750 наносекунд. Скорость увеличивается по мере роста напряжения и составляет 250 наносекунд при десяти вольтах и 190 наносекунд — при 15 вольтах. Однако по этому показателю устройство

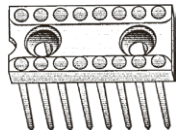
на основе микросхем КМОП сильно отстает от 4-битного ТТЛ-сумматора, время установки которого 24 наносекунды. (Двадцать пять лет назад компромисс между скоростью микросхемы ТТЛ и низким энергопотреблением микросхемы КМОП был довольно явным. Сегодня существуют версии ТТЛ-чипов с малым энергопотреблением и высокоскоростные версии микросхем КМОП.)

На практике соединение микросхем начинается на пластиковой *макетной плате*.



Каждые пять отверстий электрически соединены под пластмассовым основанием. Микросхема вставляется в макетную плату так, чтобы она опиралась на длинную центральную борозду, а ее выводы попадали в отверстия по обе стороны. Каждый вывод ИС при этом электрически совмещается с четырьмя другими отверстиями. Микросхемы объединяются с помощью проводов, вставляемых в другие отверстия.

Вы можете обеспечить постоянное соединение микросхем, используя технологию под названием *монтаж накруткой*. В данном случае каждая микросхема вставляется в гнездо с длинными квадратными штырьками.



Каждый штырек соответствует выходу микросхемы. Сами гнезда располагаются в тонких перфорированных платах. С обратной стороны платы вы используете специальный моточный агрегат для того, чтобы плотно обмотать штырек тонким изолированным проводом. Острые края штырька прорывают изоляцию, благодаря чему между штырьком и проводом возникает электрическое соединение.

Если бы вы занимались производством конкретного устройства на основе ИС, вероятно, использовали бы *печатную плату*. В былые времена ее мог изготовить даже любитель. Плата — это пластина с отверстиями, покрытая

тонким слоем медной фольги. Те участки фольги, которые требуется сохранить, покрываются кислотостойким веществом, после чего остальная часть протравливается кислотой. Затем вы можете припаять гнезда ИС (или сами ИС) непосредственно к медному покрытию. Однако из-за большого количества взаимосвязей между ИС оставшейся области медной фольги обычно оказывается недостаточно, поэтому изготавливаемые промышленным способом печатные платы имеют несколько уровней межсоединений.

К началу 1970-х стало возможным использовать ИС для сборки компьютерного процессора целиком на единой плате. А размещение всего процессора в одном чипе было лишь вопросом времени. Несмотря на то что компания Texas Instruments запатентовала однокристальный компьютер в 1971 году, честь его создания принадлежит компании Intel, основанной в 1968 году бывшими сотрудниками Fairchild Semiconductors Робертом Нойсом и Гордоном Муром. Первым важным продуктом компании Intel в 1970 году стал чип памяти с наибольшей на тот момент емкостью 1024 бит.

Компания Intel занималась разработкой микросхем для программируемого калькулятора, который собиралась производить японская Busicom, когда ее инженеры решили использовать другой подход. Как отметил инженер Intel Тед Хофф, «вместо калькулятора с возможностью программирования я хотел создать компьютер общего назначения, запрограммированный на выполнение функций калькулятора». Это привело к разработке Intel 4004, первого «компьютера в чипе», или *микропроцессора*. Продажи микросхемы 4004 начались в ноябре 1971 года, она содержала 2300 транзисторов. (Согласно закону Мура, микропроцессоры, созданные 18 лет спустя, должны содержать примерно в 4000 раз больше транзисторов, или около десяти миллионов. Это предсказание оказалось довольно точным.)

Теперь, когда вам известно количество транзисторов в микросхеме 4004, опишу еще три важные характеристики.

Во-первых, схема 4004 — это 4-разрядный микропроцессор, следовательно, ширина шины данных процессора составляла всего четыре бита. При сложении или вычитании чисел он был способен обрабатывать только четыре бита за один такт. Напротив, компьютер, разработанный в главе 17, имеет 8-разрядные шины данных и является 8-разрядным процессором. Как мы вскоре увидим, 8-разрядные микропроцессоры быстро превзошли 4-разрядные. Однако на этом никто не остановился. В конце 1970-х годов появились 16-разрядные микропроцессоры. Если вы вспомните компьютер из главы 17 и несколько команд, необходимых для сложения двух 16-разрядных чисел с помощью 8-разрядного процессора, то оцените преимущество 16-разрядного процессора. В середине 1980-х были разработаны 32-разрядные микропроцессоры, которые с тех пор остаются стандартом для домашних компьютеров.

Во-вторых, максимальная *тактовая частота* микросхемы 4004 составляла 108 тысяч циклов в секунду, или 108 килогерц. Тактовая частота — это максимальная скорость тактового генератора, который можно подключить к микропроцессору, чтобы заставить его работать. Более высокая скорость может привести к некорректной работе. К 1999 году тактовая частота микропроцессоров, предназначенных для домашних компьютеров, достигла отметки 500 мегагерц, что примерно в 5000 раз быстрее по сравнению с аналогичным показателем микросхемы 4004.

В-третьих, объем *адресуемой памяти* микросхемы 4004 составлял 640 байт. Хотя это значение кажется смехотворно маленьким, оно соответствовало емкости микросхем памяти того времени. Как вы увидите далее, уже спустя два года микропроцессоры могли обращаться к 64 килобайтам памяти, что соответствует возможностям компьютера из главы 17. В 1999 году микропроцессоры Intel могли обращаться к 64 терабайтам памяти, однако это слишком много, учитывая, что оперативная память большинства домашних компьютеров не превышала 256 мегабайт.

Эти три показателя ничего не говорят о *возможностях* компьютера. Например, 4-разрядный процессор может складывать 32-разрядные числа, просто разделяя их на 4-разрядные фрагменты. В некотором отношении все цифровые компьютеры одинаковы. Если аппаратное обеспечение одного процессора может решить задачу, которая не под силу другому, третий способен справиться с задачей с помощью программного обеспечения; в конечном итоге все они делают одно и то же. Именно это подразумевалось в статье Алана Тьюринга 1937 года о вычислимости.

Быстродействие — одна из самых важных причин, по которой мы вообще используем компьютеры. Максимальная тактовая частота оказывает очевидное влияние на общую скорость работы процессора, поскольку определяет быстроту выполнения каждой команды. Ширина шины данных процессора также влияет на его быстродействие. Несмотря на то что 4-разрядный процессор способен складывать 32-разрядные числа, при решении этой задачи он сильно уступает 32-разрядному. Тем не менее вы можете не сразу осознать, какое влияние на быстродействие оказывает максимальный объем адресуемой памяти процессора. Поначалу может показаться, что адресуемая память не имеет ничего общего со скоростью работы, а вместо этого указывает на ограничение способности процессора выполнять определенные функции, которые могут потребовать большого объема памяти. Однако процессор всегда может обойти этот предел, используя некоторые адреса памяти, чтобы управлять другими средствами для сохранения и извлечения информации. (Представьте, что каждый байт, записанный в конкретную ячейку памяти, — это фактически отверстие,

пробитое в бумажной ленте, а каждый байт, считанный из этой ячейки, считывается с ленты.) Однако этот процесс замедляет работу всего компьютера. Снова проблема в быстродействии. Конечно, эти три показателя лишь примерно отражают скорость работы микропроцессора. Они ничего не сообщают о внутренней архитектуре микропроцессора или об эффективности и возможностях команд, написанных на машинном языке. По мере усложнения процессоров многие распространенные задачи, ранее выполнявшиеся программным обеспечением, включались в функционал самого процессора. В следующих главах приведу примеры, отражающие эту тенденцию.

Все цифровые компьютеры обладают одинаковыми возможностями. Они не могут делать ничего, что выходило бы за пределы потенциала примитивной вычислительной машины, разработанной Аланом Тьюрингом. При этом скорость процессора, *конечно же*, влияет на общую полезность компьютерной системы. Например, любой компьютер, который считает медленнее, чем человеческий мозг, является бесполезным. И мы вряд ли сможем посмотреть фильм на экране современного компьютера, если процессору потребуется целая минута для того, чтобы нарисовать один кадр.

Вернемся в середину 1970-х. Несмотря на свои ограничения, микросхема 4004 стала началом новой эпохи. К апрелю 1972 года компания Intel выпустила микросхему 8008 — 8-разрядный микропроцессор с тактовой частотой 200 килогерц и адресуемой памятью объемом 16 килобайта. (Видите, как легко описать процессор с помощью всего лишь трех параметров?) А в 1974 году в течение пятимесячного периода компании Intel и Motorola выпустили микропроцессоры, превосходящие по своим характеристикам микросхему 8008. Эти два микропроцессора изменили мир.

Глава 19
Два классических микропроцессора

Микропроцессор, представляющий собой совокупность всех компонентов центрального процессорного устройства (ЦПУ), объединенных на одной кремниевой микросхеме, появился в 1971 году. Начало было скромным: первый микропроцессор Intel 4004 содержал около 2300 транзисторов. Сегодня количество транзисторов в микропроцессорах, предназначенных для домашних компьютеров, приблизилось к отметке в десять миллионов.

Тем не менее принцип его работы на фундаментальном уровне остался прежним. Огромное количество дополнительных транзисторов в современных чипах может выполнять интересные функции; на первых этапах изучения оно скорее отвлекает, чем проясняет ситуацию. Для четкого представления работы микропроцессора рассмотрим его первые версии.

Микропроцессоры, о которых пойдет речь, появились в 1974 году. В апреле Intel представила свою микросхему 8080, а в августе Motorola, которая с начала 1950-х годов занималась производством полупроводников и транзисторных устройств, выпустила микросхему 6800. В тот год на рынке появились не только эти микропроцессоры. Тогда же компания Texas Instruments представила свою 4-разрядную микросхему TMS1000, которая устанавливалась во многие калькуляторы, игрушки и другие устройства, а National Semiconductor выпустила первый 16-разрядный микропроцессор PACE. Оглядываясь назад, можно сказать, что микросхемы 8080 и 6800, безусловно, сыграли наиболее значимую роль в истории развития вычислительной техники.

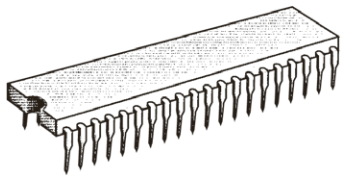
Компания Intel начала продавать микросхему 8080 по цене 360 долларов США, как бы подтрунивая над мейнфреймами System/360 компании IBM, на которые крупные корпорации тратили миллионы долларов. Нельзя сказать, что микросхема 8080 могла тягаться с мейнфреймом System/360, однако в последующие несколько лет IBM не могла обойти вниманием эти миниатюрные компьютеры.

Чип 8080 — это 8-разрядный микропроцессор, который содержит около 6000 транзисторов, работает с тактовой частотой два мегагерца и может адресовать 64 килобайта памяти. Микросхема 6800 содержит около 4000 транзисторов и тоже может адресовать 64 килобайта памяти. Первые чипы 6800 работали на частоте один мегагерц, но к 1977 году Motorola представила их усовершенствованные версии, работавшие с частотой 1,5 и 2 мегагерца.

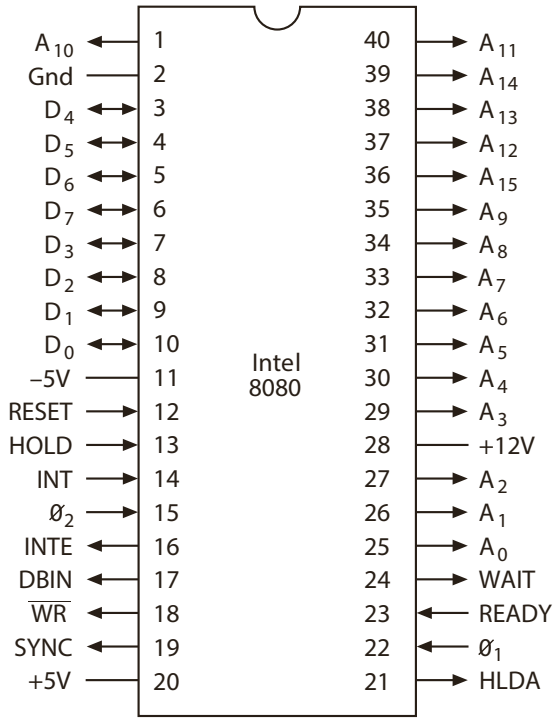
Микросхемы 8080 и 6800 называются *однокристальными* микропроцессорами, или *однокристальными компьютерами*. Процессор — это только один из компонентов компьютера. В дополнение к нему требуются по крайней мере некоторое оперативное запоминающее устройство (ОЗУ), какой-то способ, позволяющий пользователю записать информацию в компьютер (устройство ввода), а также извлечь ее из него (устройство вывода). Кроме того, необходимо еще несколько других микросхем для объединения всех компонентов. Я опишу их подробнее в главе 21.

А сейчас рассмотрим сам микропроцессор. Часто его описание сопровождается блок-схемой, которая иллюстрирует внутренние компоненты микропроцессора и то, как они связаны между собой. Однако на это мы уже насмотрелись в главе 17. Здесь мы постараемся понять, что происходит внутри процессора, наблюдая, как он взаимодействует с внешним миром. Другими словами, мы можем представить микропроцессор в качестве «черного ящика», внутреннюю работу которого нам не обязательно досконально изучать, чтобы разобраться в его функциях. Для этого достаточно ознакомиться с входными и выходными сигналами, в частности набором команд.

Чипы 8080 и 6800 — это интегральные микросхемы с 40 выводами. Чаще всего длина их корпуса составляет пять сантиметров, ширина — около 1,5 сантиметра, а высота — около трех миллиметров.



Разумеется, мы говорим только о корпусе. Размер кремниевой пластины внутри намного меньше: в ранних версиях 8-разрядных микропроцессоров это квадрат со стороной около шести миллиметров. Корпус защищает кремниевый чип и обеспечивает доступ ко всем его входам и выходам. На следующей схеме показаны функции 40 выводов микросхемы 8080.



Каждому из созданных нами электрических или электронных устройств нужен источник питания. Одна из особенностей микросхемы 8080 — ей необходимы *три* различных напряжения. На контакт 20 должно подаваться напряжение пять вольт, на контакт 11 — минус пять вольт, а на контакт 28 — 12 вольт. Контакт 2 подключается к земле. (В 1976 году Intel выпустила микросхему 8085 с более простой системой электропитания.)

Все остальные контакты изображены в виде стрелок. Стрелка, направленная *от* микросхемы, обозначает *выходной* сигнал. Это контролируемый микропроцессором сигнал, на который реагируют другие чипы. Стрелка, направленная *к* микросхеме, — *входной* сигнал, поступающий от другого чипа. Некоторые контакты используются в качестве как входов, так и выходов.

Для работы процессора, описанного в главе 17, необходим осциллятор. Микросхеме 8080 требуются два разных сигнала тактовой синхронизации с частотой два мегагерца, обозначенные на схеме символами $Ø_1$ и $Ø_2$ рядом с контактами 22 и 15. Эти сигналы удобнее генерировать с помощью другого чипа компании Intel под названием генератор тактовых импульсов 8224. К этому чипу просто подключается кристалл кварца с частотой 18 мегагерц, а все остальное делает генератор.

Микропроцессор всегда предусматривает несколько выходных сигналов для адресации памяти. Количество сигналов, используемых для этой цели, напрямую связано с объемом памяти, к которой может обращаться микропроцессор. Чип 8080 имеет 16 таких сигналов, обозначенных символами от A_0 до A_{15}, что позволяет ему адресовать 2^{16}, или 65 536 байт.

Микропроцессор 8080 является 8-разрядным, значит, он считывает данные из памяти и записывает их по восемь бит за один раз. Для этого используются контакты с D_0 по D_7, которые работают как на вход, так и на выход. Когда микропроцессор считывает байт из памяти, эти контакты функционируют как входы; когда микропроцессор записывает байт в память — как выходы.

Еще десять контактов микропроцессора предусмотрены для *управляющих* сигналов. Например, для сброса микропроцессора используется входной сигнал RESET. Выходной сигнал WR заставляет микропроцессор записать байт в память. (Сигнал WR соответствует входному сигналу записи памяти W.) Кроме того, иногда при считывании микропроцессором команд из памяти на контакты с D_0 по D_7 подаются другие управляющие сигналы. Для обработки этих дополнительных сигналов компьютеры, построенные на основе микропроцессора 8080, обычно используют системный контроллер 8228. Система управляющих сигналов микропроцессора 8080 печально известна своей сложностью, поэтому, если вы не намерены собирать компьютер на основе этого чипа, лучше не мучайте себя, пытаясь в ней разобраться.

Предположим, что микропроцессор 8080 подключен к памяти объемом 64 килобайта, с которой мы можем обмениваться данными независимо от микропроцессора.

После сброса микропроцессор 8080 считывает байт, хранящийся в ячейке памяти 0000h. Для этого на адресные контакты с A_0 по A_{15} подаются 16 нулей. Байт, который он считывает, должен соответствовать команде процессора 8080, а сам процесс его считывания называется *выборкой команды*.

В компьютере, который мы собрали в главе 17, все команды, за исключением HLT, занимали в памяти по три байта, один байт содержал код команды, а в двух других хранился адрес. Команды процессора 8080 могут занимать один, два или три байта. Одни команды заставляют процессор 8080 считать байт из определенной ячейки памяти, другие — записать байт в определенную ячейку, третьи — выполнить некие внутренние операции без использования оперативной памяти. После обработки первой команды процессор 8080 обращается ко второй команде, хранящейся в памяти, и т. д. Совокупность этих команд — компьютерная программа, способная выполнять интересные функции.

Когда процессор 8080 работает с максимальной частотой два мегагерца, каждый тактовый цикл длится 500 наносекунд (1 / 2 000 000 циклов в секунду = = 0,000000500 секунды). Команды компьютера, описанного в главе 17, выполнялись за четыре тактовых цикла. На исполнение команд для процессора 8080 требуется от 4 до 18 тактовых циклов, то есть от двух до девяти микросекунд (миллионных долей секунды).

Вероятно, чтобы понять, на что способен конкретный микропроцессор, нужно изучить весь набор его команд.

Итоговая модель компьютера из главы 17 выполняла всего 12 команд. Набор 8-разрядного микропроцессора может легко содержать до 256 команд, при этом код каждой команды соответствует конкретному 8-битному значению. (На самом деле команд может быть даже больше, если некоторым из них будет соответствовать 2-байтный код.) Микропроцессор 8080 не заходит так далеко, но его набор включает 244 кода. Это количество может показаться довольно большим, однако в целом процессор 8080 не сильно превосходит компьютер из главы 17. Например, если вам нужно произвести операцию умножения или деления с помощью процессора 8080, все равно придется написать для этого небольшую программу.

Как вы помните из главы 17, каждый код в наборе команд процессора обычно соответствует определенному мнемокоду, и некоторые из них сопровождаются аргументами. Однако эти мнемокоды служат исключительно для удобства при обращении к кодам команд. Процессор считывает только байты; он ничего не знает о соответствующем им тексте. (Для ясности буду несколько вольно обращаться с мнемокодами из документации к процессору Intel 8080.)

Набор компьютера из главы 17 включал две важные команды, которые мы первоначально назвали «Загрузить» и «Сохранить». Каждая из этих команд занимала три байта памяти. Первый байт команды «Загрузить» соответствовал ее коду, а два следующих — 16-битному адресу. Процессор загружал байт, хранящийся по этому адресу, в аккумулятор. Аналогично команда «Сохранить» консервировала содержимое аккумулятора по указанному адресу.

Затем мы обнаружили, что коды этих двух команд можно сократить, используя мнемокоды.

```
LOD A,[aaaa]
STO [aaaa], A
```

Здесь *A* — аккумулятор (место назначения для команды «Загрузить» и источник для команды «Сохранить»), а фрагмент *aaaa* — 16-битный адрес в памяти, который обычно записывается с помощью четырех шестнадцатеричных цифр.

Восьмиразрядный аккумулятор в процессоре 8080 обозначается буквой *A*, как и аккумулятор в главе 17. Кроме того, в наборе этого процессора есть две команды, которые делают то же самое, что и команды «Загрузить» и «Сохранить» из главы 17. В процессоре 8080 этим командам соответствуют коды 32h и 3Ah, за каждым из которых следует 16-битный адрес, а мнемокодами для них являются STA (Store Accumulator — сохранить из аккумулятора) и LDA (Load Accumulator — загрузить в аккумулятор).

Код	Команда
32	`STA [aaaa], A`
3A	`LDA A,[aaaa]`

В дополнение к аккумулятору процессор 8080 имеет шесть *регистров*, которые также могут содержать 8-битные значения. Эти регистры похожи на аккумулятор. На самом деле аккумулятор считается регистром особого типа. Подобно аккумулятору, остальные шесть регистров являются защелками; процессор может перемещать байты из памяти в регистры и из регистров обратно в память. В отличие от аккумулятора, регистры не так функциональны. Например, при сложении двух 8-битных чисел результат всегда попадает в аккумулятор, а не в один из регистров.

Шесть дополнительных регистров в процессоре 8080 называются B, C, D, E, H и L. Первым делом люди спрашивают: «Что же случилось с F и G?» — а затем задают второй вопрос: «A как насчет I, J и K?» Ответ заключается в том, что регистры H и L обладают некоторыми особенностями, а их название происходит от слов high и low. Часто 8-битные значения в регистрах H и L обрабатываются вместе в виде 16-битной *пары регистров* HL, при этом в регистре H содержится старший (high) байт, а в регистре L — младший (low). Это 16-битное значение используется для адресации памяти. Чуть позже мы увидим, как это работает.

Так ли необходимы все эти регистры? Почему мы в них не нуждались, собирая компьютер в главе 17? Теоретически использовать их не обязательно, а практически — весьма удобно. Многие компьютерные программы способны одновременно манипулировать несколькими числами. Это проще всего делать, если числа хранятся не в памяти, а в регистрах микропроцессора. Кроме того, программа работает быстрее: чем реже она обращается к памяти, тем быстрее выполняется.

Для команды процессора 8080 под названием MOV (Move — переместить) предусмотрено 63 кода. Эти коды занимают только один байт и, как правило, перемещают содержимое одного регистра в другой или в тот же самый. Множество команд MOV — следствие использования в микропроцессоре семи регистров (включая аккумулятор).

Вот первые 32 команды MOV. Помните, что место назначения соответствует аргументу слева, а источник — аргументу справа.

Код	Команда	Код	Команда
40	MOV B, B	50	MOV D, B
41	MOV B, C	51	MOV D, C
42	MOV B, D	52	MOV D, D
43	MOV B, E	53	MOV D, E
44	MOV B, H	54	MOV D, H
45	MOV B, L	55	MOV D, L
46	MOV B,[HL]	56	MOV D,[HL]
47	MOV B, A	57	MOV D, A
48	MOV C, B	58	MOV E, B
49	MOV C, C	59	MOV E, C
4A	MOV C, D	5A	MOV E, D
4B	MOV C, E	5B	MOV E, E
4C	MOV C, H	5C	MOV E, H
4D	MOV C, L	5D	MOV E, L
4E	MOV C,[HL]	5E	MOV E,[HL]
4F	MOV C, A	5F	MOV E, A

Как видите, весьма удобные команды. При наличии значения в одном из регистров можно переместить его в другой. Обратите внимание на четыре команды, которые используют пару регистров HL, например на следующую.

```
MOV B,[HL]
```

Упомянутая выше команда LDA перемещает байт из памяти в аккумулятор; 16-битный адрес этого байта следует непосредственно за кодом команды LDA. Эта команда MOV перемещает байт из памяти в регистр B. Однако адрес байта, который должен быть загружен в регистр, хранится в паре регистров HL. Как 16-битный адрес оказался в паре регистров HL? Это могло произойти разными способами. Возможно, этот адрес был каким-то образом вычислен.

В общем, обе команды загружают байт из памяти в микропроцессор, но используют два разных метода для адресации памяти. Первый метод называется *прямой адресацией*, а второй — *индексной адресацией*.

```
LDA A,[aaaa]
MOV B,[HL]
```

Второй набор из 32 команд MOV показывает, что ячейки памяти, адресуемые парой регистров HL, могут являться не только источником, но и местом назначения.

Код	Команда	Код	Команда
40	MOV B, B	50	MOV D, B
60	MOV H, B	70	MOV [HL], B
61	MOV H, C	71	MOV [HL], C
62	MOV H, D	72	MOV [HL], D
63	MOV H, E	73	MOV [HL], E
64	MOV H, H	74	MOV [HL], H
65	MOV H, L	75	MOV [HL], L
66	MOV H,[HL]	76	HLT
67	MOV H, A	77	MOV [HL], A
68	MOV L, B	78	MOV A, B
69	MOV L, C	79	MOV A, C
6A	MOV L, D	7A	MOV A, D
6B	MOV L, E	7B	MOV A, E
6C	MOV L, H	7C	MOV A, H
6D	MOV L, L	7D	MOV A, L
6E	MOV L,[HL]	7E	MOV A,[HL]
6F	MOV L, A	7F	MOV A, A

Некоторые из этих команд, например `MOV A, A`, не делают ничего полезного. Команды `MOV [HL],[HL]` вообще не существует. Код, который мог бы ей соответствовать, выделен команде HLT (Halt — остановить).

Более показательный способ анализа команд MOV — рассмотрение битового шаблона их кода. Код команды MOV состоит из восьми битов:

```
01ннниии,
```

где буквы *ннн* соответствуют 3-битному коду места назначения, а *иии* — 3-битному коду источника. Эти 3-битные коды обозначают следующие регистры.

000 = регистр B
001 = регистр C
010 = регистр D
011 = регистр E
100 = регистр H
101 = регистр L

110 = ячейка памяти по адресу HL
111 = аккумулятор

Команда `MOV L, E` соответствует коду 01101011, или 6Bh. Вы можете свериться с предыдущей таблицей, чтобы убедиться в этом.

Вероятно, где-то внутри процессора 8080 три бита *иии* используются в селекторе «8 на 1», а три бита *ннн* управляют дешифратором «3 на 8», определяющим регистр, где будет зафиксировано значение.

Регистры B и C также можно использовать как 16-битную пару регистров BC, а регистры D и E — как 16-битную пару регистров DE. Если в любой из этих пар регистров содержится адрес ячейки памяти, откуда вы хотите считать или куда хотите записать байт, можете использовать следующие команды.

Код	Команда	Код	Команда
02	`STAX [BC], A`	0A	`LDAX A,[BC]`
12	`STAX [DE], A`	1A	`LDAX A,[DE]`

Другой тип команды Move называется Move Immediate («Переместить непосредственно») и обозначается мнемокодом MVI. Эта команда состоит из двух байтов. Первый — код команды, второй — байт данных. Этот байт перемещается из памяти в один из регистров или в ячейку памяти, адрес которой содержится в паре регистров HL.

Код	Команда
06	`MVI B, xx`
0E	`MVI C, xx`
16	`MVI D, xx`
1E	`MVI E, xx`
26	`MVI H, xx`
2E	`MVI L, xx`
36	`MVI [HL], xx`
3E	`MVI A, xx`

Например, после выполнения команды `MVI E,37h` в регистре E будет содержаться байт 37h. Этот третий метод обращения к памяти называется *непосредственной адресацией*.

Набор из 32 кодов команд выполняет четыре основные арифметические операции, с которыми мы познакомились, когда собирали процессор (глава 17). К ним относятся сложение (ADD), сложение с переносом (ADC),

вычитание (SUB) и вычитание с заимствованием (SBB). Во всех случаях аккумулятор является одним из двух операндов, а также местом назначения для результата.

Код	Команда	Код	Команда
80	ADD A, B	90	SUB A, B
81	ADD A, C	91	SUB A, C
82	ADD A, D	92	SUB A, D
83	ADD A, E	93	SUB A, E
84	ADD A, H	94	SUB A, H
85	ADD A, L	95	SUB A, L
86	ADD A,[HL]	96	SUB A,[HL]
87	ADD A, A	97	SUB A, A
88	ADC A, B	98	SBB A, B
89	ADC A, C	99	SBB A, C
8A	ADC A, D	9A	SBB A, D
8B	ADC A, E	9B	SBB A, E
8C	ADC A, H	9C	SBB A, H
8D	ADC A, L	9D	SBB A, L
8E	ADC A,[HL]	9E	SBB A,[HL]
8F	ADC A, A	9F	SBB A, A

Предположим, что в аккумуляторе A содержится байт 35h, а в регистре B — байт 22h. После выполнения команды `SUB A, B` в аккумуляторе будет содержаться байт 13h.

Если в A содержится байт 35h, в регистре H — 10h, в регистре L — 7Ch, в ячейке памяти 107Ch — 4Ah, то при выполнении команды `ADD A,[HL]` байт в аккумуляторе (35h) прибавляется к байту в ячейке, к которой обращается пара регистров HL (4Ah), а результат (7Fh) сохраняется в аккумуляторе.

Команды ADC и SBB позволяют процессору 8080 складывать и вычитать 16-, 24-, 32-битные числа, а также числа большей разрядности. Предположим, что пары регистров BC и DE содержат 16-битные числа. Вы хотите сложить их и поместить результат в пару регистров BC. Это можно сделать так.

```
MOV A, C  ; младший байт
ADD A, E
MOV C, A
MOV A, B  ; старший байт
ADC A, D
MOV B, A
```

Для сложения используются две команды: ADD — для младшего байта, ADC — для старшего. Любой бит переноса, возникающий в результате первого сложения, участвует во втором сложении. Поскольку прибавлять можно только к значению в аккумуляторе, в этом небольшом фрагменте кода команда MOV используется не менее четырех раз. Команды MOV очень часто встречаются в программном коде для процессора 8080.

Пришло время поговорить о флагах микросхемы 8080. В процессоре из главы 17 использовались флаг переноса и флаг нуля. Микросхема 8080 предусматривает еще три флага: знака, четности и вспомогательного переноса. Все флаги хранятся в 8-битном регистре, который называется *словом состояния программы* (Program Status Word, PSW). Такие команды, как LDA, STA или MOV, не влияют на эти флаги. Однако команды ADD, SUB, ADC и SBB изменяют флаги следующим образом:

— флаг знака устанавливается в 1, если старший бит результата равен 1, то есть если результат отрицательный;
— флаг нуля устанавливается в 1, если результат равен 0;
— флаг четности устанавливается в 1, если результат *четен*, то есть выраженный в двоичном формате результат содержит четное количество 1; флаг четности устанавливается в 0, если результат *нечетен*; флаг четности иногда используется для грубой проверки результата на наличие ошибок; при написании программ для процессора 8080 этот флаг используется редко;
— флаг переноса устанавливается в 1, если в результате выполнения команды ADD или ADC возникает бит переноса либо в результате выполнения команд SUB и SBB бит переноса не возникает (такая реализация флага переноса отличается от того, как он был реализован в компьютере из главы 17);
— флаг вспомогательного переноса устанавливается в 1, если в результате выполнения команды возникает перенос из младшей тетрады в старшую; этот флаг используется только для команды DAA (Decimal Adjust Accumulator — десятичная коррекция аккумулятора).

На флаг переноса непосредственно влияют две команды.

Код	Команда	Значение
37	STC	Установить флаг переноса в 1
3F	CMC	Дополнить флаг переноса до 1 или инвертировать флаг переноса

В отличие от компьютера из главы 17, который тоже выполнял команды ADD, ADC, SUB и SBB (хотя и не с такой же степенью гибкости), процессор 8080 способен еще и на булевы операции И, ИЛИ и исключающее ИЛИ. За выполнение арифметических и логических операций отвечает арифметико-логическое устройство процессора.

Код	Команда	Код	Команда
A0	AND A, B	B0	OR A, B
A1	AND A, C	B1	OR A, C
A2	AND A, D	B2	OR A, D
A3	AND A, E	B3	OR A, E
A4	AND A, H	B4	OR A, H
A5	AND A, L	B5	OR A, L
A6	AND A,[HL]	B6	OR A,[HL]
A7	AND A, A	B7	OR A, A
A8	XOR A, B	B8	CMP A, B
A9	XOR A, C	B9	CMP A, C
AA	XOR A, D	BA	CMP A, D
AB	XOR A, E	BB	CMP A, E
AC	XOR A, H	BC	CMP A, H
AD	XOR A, L	BD	CMP A, L
AE	XOR A,[HL]	BE	CMP A,[HL]
AF	XOR A, A	BF	CMP A, A

Команды AND, OR и XOR выполняются *побитово*, то есть отдельно над каждой парой битов. Например, в результате выполнения следующих команд значение в аккумуляторе будет равно 05h.

```
MVI A,0Fh
MVI B,55h
AND A, B
```

Если бы последней была команда OR, то результат был бы равен 5Fh; если бы последней была команда XOR — 5Ah.

Команда CMP (Compare — сравнить) аналогична команде SUB, за исключением того, что результат не сохраняется в аккумуляторе. Другими словами, команда CMP выполняет вычитание, а затем удаляет результат. В чем же смысл? Во флагах! Флаги говорят о том, как два сравниваемых байта соотносятся. Рассмотрим, например, следующие команды.

```
MVI B,25h
CMP A,B
```

После их выполнения содержимое аккумулятора (A) остается прежним. Если значение в A равно 25h, будет установлен флаг нуля, а если значение в A меньше 25h — флаг переноса.

Для восьми арифметических и логических операций также существуют версии, которые выполняются непосредственно над байтами.

Код	Команда	Код	Команда
C6	ADI A, xx	E6	ANI A, xx
CE	ACI A, xx	EE	XRI A, xx
D6	SUI A, xx	F6	ORI A, xx
DE	SBI A, xx	FE	CPI A, xx

Например, две приведенные выше строки можно заменить следующей.

```
CPI A,25h
```

Вот еще две команды для процессора 8080.

Код	Команда
27	DAA
2F	CMA

Команда CMA (Complement Accumulator — дополнить аккумулятор) выполняет дополнение значения в аккумуляторе до 1. Каждый 0 обращается в 1, а 1 — в 0. Если в аккумуляторе содержится значение 01100101, то после исполнения команды CMA в нем будет содержаться значение 10011010. Этого же результата можно достичь и с помощью следующей команды.

```
XRI A, FFh
```

Упомянутая выше команда DAA (Decimal Adjust Accumulator — десятичная коррекция аккумулятора), вероятно, является самой сложной в наборе команд процессора 8080. Специально для нее в микропроцессоре предусмотрено небольшое устройство.

DAA помогает программисту выполнять арифметические операции над десятичными числами в кодировке BCD (binary-coded decimal — десятичное в двоичной кодировке), где каждая тетрада может принимать значение только

в диапазоне от 0000 до 1001, то есть от 0 до 9 в десятичном выражении. В формате BCD в восьми битах байта могут храниться две десятичные цифры.

Предположим, что в аккумуляторе содержится BCD-значение 27h, которое фактически соответствует десятичному значению 27, а в регистре B содержится BCD-значение 94h. (Обычно шестнадцатеричное значение 27h эквивалентно десятичному значению 39.) В результате выполнения следующих команд в аккумуляторе будет содержаться значение BBh, которое, разумеется, не является BCD-значением, поскольку в кодировке BCD-значение тетрады не может превышать 9.

```
MVI A,27h
MVI B,94h
ADD A, B
```

Однако при выполнении команды DAA в аккумулятор помещается значение 21h и устанавливается флаг переноса, поскольку сумма десятичных чисел 27 и 94 равна 121. Эта команда может пригодиться для арифметических операций над числами в кодировке BCD.

Часто возникает необходимость в прибавлении 1 к значению или в вычитании 1 из значения. В программе для выполнения умножения, описанной в главе 17, нужно вычесть из значения 1, и мы делали это, прибавляя значение FFh, которое является дополнением до 2 числа –1. Процессор 8080 предусматривает специальные команды для увеличения на 1 (инкрементирования) и уменьшения на 1 (декрементирования) значения в регистре или в ячейке памяти.

Код	Команда	Код	Команда
04	INR B	05	DCR B
0C	INR C	0D	DCR C
14	INR D	15	DCR D
1C	INR E	1D	DCR E
24	INR H	25	DCR H
2C	INR L	2D	DCR L
34	INR [HL]	35	DCR [HL]
3C	INR A	3D	DCR A

Однобайтовые команды INR и DCR влияют на все флаги, кроме флага переноса.

Набор команд процессора 8080 также включает четыре команды *циклического сдвига*, которые сдвигают содержимое аккумулятора на один бит влево или вправо.

Код	Команда	Значение
07	RLC	Сдвинуть аккумулятор влево
0F	RRC	Сдвинуть аккумулятор вправо
17	RAL	Сдвинуть аккумулятор влево через бит переноса
1F	RAR	Сдвинуть аккумулятор вправо через бит переноса

Эти команды влияют только на флаг переноса.

Предположим, что аккумулятор содержит значение A7h, или 10100111 в двоичном формате. Команда RLC сдвигает биты влево. Старший бит, выталкиваемый за левую границу разрядной сетки, становится младшим, а также определяет состояние флага переноса. В результате получается значение 01001111, а флаг переноса устанавливается в 1. Команда RRC точно так же сдвигает биты вправо. После выполнения команды RRC значение 10100111 превращается в 11010011, а флаг переноса опять устанавливается в 1.

Команды RAL и RAR работают несколько иначе. При выполнении команды RAL содержимое аккумулятора сдвигается влево, старший бит сохраняется во флаге переноса, а в младший бит записывается предыдущее значение флага переноса. Например, если аккумулятор содержит значение 10100111, а флаг переноса равен 0, то после выполнения команды RAL содержимое аккумулятора меняется на 01001110, а во флаг переноса записывается 1. При тех же начальных условиях после выполнения команды RAR значение аккумулятора аналогично меняется на 01010011, а во флаге переноса сохраняется значение 1.

Команды сдвига удобны при умножении числа на 2 (сдвиг влево) и при делении числа на 2 (сдвиг вправо).

Память, к которой обращается микропроцессор, называется памятью с *произвольным доступом* потому, что микропроцессор может получить доступ к любой конкретной ячейке, просто предоставив ее адрес. Память RAM скорее напоминает книгу, которую можно открыть на любой странице, чем недельную подшивку газет на микрофильме. Чтобы найти нужную информацию в субботнем выпуске, мы должны просмотреть бóльшую часть газет. Так, для воспроизведения последней песни на кассете мы должны практически полностью перемотать одну из ее сторон. Микрофильм и магнитная лента относятся к запоминающим устройствам не с произвольным, а с *последовательным доступом*.

Память с произвольным доступом, безусловно, хороша, особенно для микропроцессоров, но иногда удобнее использовать запоминающее устройство,

доступ к которому осуществляется непроизвольно и непоследовательно. Допустим, вы работаете в офисе, и сотрудники подходят к вашему столу, чтобы дать задание. Выполнение каждого из них предполагает использование папки с документами. Часто при работе над одним заданием вы обнаруживаете, что не можете продолжать, пока не выполните определенную задачу, используя другую папку. Так что поверх первой папки вы кладете вторую и работаете с ней. Затем вам дают еще одно задание, более приоритетное, чем предыдущее, и вы кладете новую папку поверх двух других. Для выполнения этого вам требуется еще одна папка с документами. И вот на вашем столе уже целая стопка из четырех папок.

Это упорядоченный способ хранения и отслеживания всех выполняемых заданий. Самая верхняя папка всегда соответствует приоритетной задаче. После окончания работы с этой папкой вы переходите к следующей. Когда наконец вы разберетесь с последней папкой на своем столе (с той, с которой начали), сможете отправиться домой.

Технически такая форма хранения данных называется «*стек*». Строится он снизу вверх, а разбирается сверху вниз. Элементы стека организованы по принципу «последним вошел — первым вышел» (Last In First Out, LIFO). Последний элемент, помещенный в стек, удаляется из него первым. Первый добавленный в стек элемент будет удален из него последним.

Компьютеры также могут использовать стек, но не для хранения заданий, а для хранения чисел, что удобно. Добавление элемента в стек называется вталкиванием (push), а его удаление — выталкиванием (pop).

Предположим, вы пишете программу на языке ассемблера, в которой используются регистры A, B и C. На каком-то этапе программе требуется выполнить еще один небольшой расчет, также предполагающий применение регистров A, B и C. В итоге нужно вернуться к тому, что вы делали раньше, и продолжить использовать регистры A, B и C с теми значениями, которые в них хранились.

Безусловно, вы можете просто сохранить значения регистров A, B и C в других ячейках памяти, а затем загрузить их оттуда обратно. Однако тогда нужно будет следить за содержимым ячеек памяти. Более удобный способ — помещение (вталкивание) значений регистров в стек.

```
PUSH A
PUSH B
PUSH C
```

Я объясню, как работают эти команды, чуть позже. Пока достаточно понять, что они каким-то образом сохраняют содержимое регистров в памяти LIFO. После выполнения этих команд ваша программа может спокойно использовать

эти регистры для других целей. Чтобы вернуть предыдущие значения, вы просто выталкиваете элементы из стека в обратном порядке.

```
POP C
POP B
POP A
```

Помните: последний помещенный в стек элемент удаляется из него в первую очередь. Случайное изменение порядка команд POP приведет к ошибке.

Преимущество стека в том, что его могут использовать разные разделы программы, не вызывая проблем. Например, после помещения в стек значений регистров A, B и C другому разделу программы может понадобиться сделать то же самое с регистрами C, D и E.

```
PUSH C
PUSH D
PUSH E
```

Для восстановления значения регистров используются команды POP.

```
POP E
POP D
POP C
```

После их выполнения из стека будут извлечены значения регистров C, B и A.

Как реализуется стек? Прежде всего, это просто раздел памяти, не используемый для хранения каких-либо других данных. Для обращения к этому разделу памяти микропроцессор 8080 предусматривает специальный 16-битный регистр, который называется *указателем стека* (Stack Pointer, SP).

Приведенные выше примеры добавления и удаления элементов из стека не вполне точно демонстрируют работу микропроцессора 8080. Команда 8080 PUSH фактически сохраняет в стеке 16-битные значения, а команда POP извлекает их. Именно поэтому вместо таких команд, как PUSH C и POP C, используем следующие восемь.

Код	Команда	Код	Команда
C5	PUSH BC	C1	POP BC
D5	PUSH DE	D1	POP DE
E5	PUSH HL	E1	POP HL
F5	PUSH PSW	F1	POP PSW

Команда PUSH BC сохраняет в стеке значения регистров B и C, а команда POP BC извлекает их. Аббревиатура PSW в последней строке означает *слова состояния программы*, которые, как вы помните, представляют собой 8-битный регистр, содержащий флаги. Две команды в нижней строке фактически помещают и извлекают из стека содержимое как аккумулятора, так и регистра PSW. Если вы хотите сохранить содержимое *всех* регистров и значения *всех* флагов, используйте следующие команды.

```
PUSH PSW
PUSH BC
PUSH DE
PUSH HL
```

Когда вам потребуется восстановить содержимое этих регистров, обращайтесь к командам POP в обратном порядке.

```
POP HL
POP DE
POP BC
POP PSW
```

Как работает стек? Предположим, что указатель стека равен 8000h. При выполнении команды PUSH BC происходит следующее:

— значение указателя стека уменьшается на 1 и становится равным 7FFFh;
— содержимое регистра B сохраняется по адресу, соответствующему значению указателя стека, то есть в ячейке 7FFFh;
— значение указателя стека уменьшается на 1 и становится равным 7FFEh;
— содержимое регистра C сохраняется по адресу, соответствующему значению указателя стека, то есть в ячейке 7FFEh.

Команда POP BC, выполняемая при значении указателя стека, все еще равном 7FFEh, производит обратные операции:

— содержимое регистра C загружается из ячейки, адрес которой соответствует значению указателя стека, то есть из ячейки 7FFEh;
— значение указателя стека увеличивается на 1 и становится равным 7FFFh;
— содержимое регистра B загружается из ячейки, адрес которой соответствует значению указателя стека, то есть из ячейки 7FFFh;
— значение указателя стека увеличивается на 1 и становится равным 8000h.

Каждая команда PUSH увеличивает размер стека на два байта. Существует вероятность того, что из-за ошибки в программе размер стека станет настолько большим, что его содержимое начнет сохраняться в ячейках, занятых необходимым программе кодом или данными. Эта проблема называется *переполнением стека*. Точно так же слишком большое количество команд POP может привести к преждевременному *исчерпанию стека*.

Если к процессору 8080 подключена память объемом 64 килобайт, имеет смысл установить начальное значение указателя стека равным 0000h. Первая команда PUSH уменьшает это значение на 1 — до FFFFh. После этого стек займет область памяти с самыми высокими адресами, которая максимально удалена от ваших программ, хранящихся, вероятно, начиная с 0000h.

Установить значение указателя стека можно с помощью команды LXI (Load Extended Immediate — расширенная непосредственная загрузка). Перечисленные далее команды также загружают в 16-битные пары регистров два байта, которые следуют за кодом команды.

Код	Команда
01	`LXI BC, xxxx`
11	`LXI DE, xxxx`
21	`LXI HL, xxxx`
31	`LXI SP, xxxx`

Команда LXI BC,527Ah эквивалентна следующим командам.

```
MVI B,52
MVI C,7Ah
```

Однако команда LXI позволяет сэкономить один байт. Кроме того, последняя команда LXI в предыдущей таблице используется для установки конкретного значения для указателя стека. Часто эта команда одной из первых выполняется микропроцессором после его перезапуска.

```
0000h: LXI SP,0000h
```

Увеличить и уменьшить на 1 значение пары регистров и указателя стека можно с помощью следующих команд.

Код	Команда	Код	Команда
03	INX BC	0B	DCX BC
13	INX DE	1B	DCX DE
23	INX HL	2B	DCX HL
33	INX SP	3B	DCX SP

Рассмотрим еще несколько 16-битных команд. Следующие команды складывают содержимое 16-битных пар регистров с содержимым пары регистров HL.

Код	Команда
09	DAD HL, BC
19	DAD HL, DE
29	DAD HL, HL
39	DAD HL, SP

Эти команды позволяют сэкономить несколько байтов. Например, первая из них, как правило, требует шесть байт.

```
MOV A, L
ADD A, C
MOV L, A
MOV A, H
ADC A, B
MOV H, A
```

Команда DAD обычно используется для вычисления адресов ячеек памяти и влияет только на флаг переноса.

Следующие два кода команд сопровождаются 2-байтовым адресом ячейки памяти и позволяют сохранить содержимое пары регистров HL в соответствующей ячейке, а также загрузить из нее содержимое в пару регистров HL.

Код	Команда	Значение
2h	SHLD [aaaa], HL	Записать число из HL в PC
2Ah	LHLD HL,[aaaa]	Загрузить данные в HL

Содержимое регистра L сохраняется по адресу *aaaa*, а содержимое регистра H — по адресу *aaaa* + 1.

Эти две команды загружают в счетчик команд (PC) или в указатель стека (SP) значение из пары регистров HL.

Код	Команда	Значение
E9h	PCHL PC, HL	Загрузить значение HL в PC
F9h	SPHL SP, HL	Загрузить значение HL в SP

Команда PCHL — своеобразная команда перехода. После нее процессор 8080 выполняет команду, код которой занимает ячейку по адресу, записанному в паре регистров HL. Команда SPHL — еще один способ установки значения указателя стека.

Следующие две команды позволяют поменять местами содержимое регистров HL с двумя байтами, являющимися «верхними» элементами стека, или с содержимым пары регистров DE.

Код	Команда	Значение
E3h	XTHL HL,[SP]	Поменять местами «верхний» элемент стека и HL
EBh	XCHG HL, DE	Поменять местами DE и HL

Из всех команд перехода для процессора 8080 пока я описал только PCHL. Как вы помните из главы 17, процессор предусматривает регистр под названием «счетчик команд», содержащий адрес ячейки памяти, из которой процессор извлекает следующую команду, подлежащую выполнению. Как правило, счетчик команд заставляет процессор выполнять команды, сохраненные в памяти, последовательно. Однако так называемые команды *перехода*, или *ветвления,* позволяют процессору отклониться от этого главного курса. Эти команды загружают в счетчик команд другое значение, поэтому следующая команда извлекается процессором из какой-то другой области памяти.

Несмотря на удобство обычной команды *перехода*, команды *условного перехода* более удобные, поскольку заставляют процессор переходить к другому адресу, основываясь на значении определенного флага, например флага переноса или флага нуля. Именно реализация условного перехода превратила автоматизированный сумматор из главы 17 в универсальный цифровой компьютер.

В процессоре 8080 имеется пять флагов, четыре из которых используются для реализации условных переходов. Набор команд 8080 содержит девять команд безусловных и условных переходов, зависящих от того, чему равны флаги нуля, переноса, четности и знака: 1 или 0.

Прежде чем продемонстрировать эти команды, хочу познакомить вас с двумя другими типами команд, имеющих отношение к переходу. Первая — CALL (вызов), она аналогична команде перехода, за исключением того, что перед загрузкой нового адреса в счетчик команд процессор сохраняет предыдущий адрес. Где он сохраняет этот адрес? Разумеется, в стеке!

Эта стратегия подразумевает, что команда вызова сохраняет информацию о том месте, *откуда был совершен переход*. Сохраненный адрес позволяет процессору вернуться в исходное местоположение. Команда для совершения обратного перехода называется RET (Return — вернуться). Она удаляет из стека 2-байтное значение и загружает его в счетчик команд.

Команды CALL и RET — чрезвычайно важные функции любого процессора, позволяющие программисту реализовывать подпрограммы, которые являются часто используемыми фрагментами кода. (Под словом «*часто*» обычно я имею в виду «более одного раза».) Подпрограммы — основные организующие элементы программ на языке ассемблера.

Обратимся к примеру. В процессе написания программы на языке ассемблера у вас возникает необходимость в перемножении двух байтов. Вы пишете код для выполнения этой операции, а затем продолжаете работу с программой. На каком-то этапе вам снова требуется перемножить два байта. Поскольку вы уже знаете, как это сделать, можно просто использовать те же команды снова и снова. Собираетесь ли вы во второй раз ввести эти команды в память? Надеюсь, что нет, поскольку это пустая трата времени и памяти. Вместо этого вам следует просто перейти к предыдущему фрагменту кода. Правда, в данном случае обычная команда перехода не сработает, поскольку она не позволяет вернуться к тому месту, с которого был совершен переход. Именно в этом случае пригодятся команды CALL и RET.

Набор команд, позволяющих перемножить два байта, идеально подходит на роль подпрограммы. Давайте рассмотрим одну из них. В главе 17 подлежащие перемножению байты и произведение хранились в определенных ячейках памяти. Приведенная далее подпрограмма 8080 умножает байт в регистре B на байт в регистре C и помещает 16-битное произведение в регистр HL.

```
Multiply: PUSH PSW          ; Сохранение изменяемых регистров
          PUSH BC

          SUB H, H          ; Установить HL (результат)
          SUB L, L          ; в 0000h

          MOV A, B          ; Сохранение множителя в A
          CPI A,00h         ; Если он равен 0,
```

```
            JZ AllDone          ; завершить программу

            MVI B,00h   ; Сохранение значения 0 в старшем байте BC
MultLoop:   DAD HL, BC          ; Сложение значений HL и BC
            DEC A               ; Уменьшение множителя на 1
            JNZ MultLoop        ; Возврат к началу цикла,если не 0

AllDone:    POP BC              ; Восстановление значений
            POP PSW             ; регистров
            RET                 ; Возврат
```

Обратите внимание: первая строка подпрограммы начинается с метки Multiply. Эта метка соответствует адресу ячейки памяти, в которой расположена подпрограмма. Подпрограмма начинается с двух команд PUSH. Как правило, она пытается сохранить (а в дальнейшем восстановить) значения любых регистров, которые могут ей потребоваться.

Затем подпрограмма записывает значение 0 в регистры H и L. Для этого вместо команды SUB можно было бы использовать команду MVI (Move Immediate — переместить непосредственно), однако в этом случае потребовались бы четыре команды, а не две. После выполнения подпрограммы в паре регистров HL будет содержаться произведение.

После этого подпрограмма перемещает содержимое регистра B (множитель) в A и проверяет, не равно ли оно 0. Если оно равно 0, подпрограмма завершается, так как произведение — 0. Поскольку значения в регистрах H и L уже равны 0, подпрограмма может просто использовать команду JZ (Jump If Zero — перейти, если ноль), чтобы перейти к двум командам POP в конце программы.

В противном случае подпрограмма записывает в регистр B значение 0. Теперь в паре регистров BC содержится 16-битное множимое, а в аккумуляторе (A) — множитель. Команда DAD прибавляет значение BC (множимое) к значению HL (произведение). Значение множителя в A уменьшается на 1. Пока он не станет равен 0, выполнение команды JNZ (Jump If Not Zero — перейти, если не ноль) будет приводить к повторному сложению значения BC со значением HL. Этот небольшой цикл будет выполняться до тех пор, пока количество операций сложения BC и HL не станет равным множителю. (Более эффективную подпрограмму для умножения можно написать, используя команды сдвига из набора команд процессора 8080.)

Эту подпрограмму для перемножения чисел, например 25h и 12h, можно использовать в программе, добавив следующий фрагмент кода.

```
MVI B,25h
MVI C,12h
CALL Multiply
```

Команда CALL сохраняет в стеке значение счетчика команд, которое представляет адрес команды, следующей после CALL. Затем CALL вызывает переход к команде, на которую указывает метка Multiply. Это начало подпрограммы. После того как подпрограмма рассчитает произведение, она выполнит команду RET, в результате чего в счетчик команд будет возвращено значение из стека. Затем будет выполнена команда, следующая после CALL.

Набор команд процессора 8080 предусматривает условные команды *вызова* и *возврата*, однако они используются реже, чем обычные команды *перехода*. Все они перечислены в следующей таблице.

Условие	Код	Команда	Код	Команда	Код	Команда
Нет	C9	RET	C3	JMP aaaa	CD	CALL aaaa
Не ноль (флаг нуля не установлен)	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
Ноль (флаг нуля установлен)	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
Нет переноса (флаг переноса не установлен)	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
Есть перенос (флаг переноса установлен)	D8	RC	DA	JC aaaa	DC	CC aaaa
Результат нечетный (флаг четности не установлен)	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
Результат четный (флаг четности установлен)	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
Результат положительный (флаг знака не установлен)	F0	RP	F2	JP aaaa	F4	CP aaaa
Результат отрицательный (флаг знака установлен)	F8	RM	FA	JM aaaa	FC	CM aaaa

Как вы знаете, к микропроцессору подключается не только память. Компьютерная система обычно требует устройства ввода и вывода (I/O), которые облегчают пользователям взаимодействие с машиной. К этим устройствам, как правило, относятся клавиатура и дисплей.

Как микропроцессор взаимодействует с этими *периферийными* устройствами? (Все подключенные к микропроцессору компоненты, кроме памяти, называются периферийными.) Конструкция периферийных устройств, подобно памяти, предусматривает интерфейс. Микропроцессор может записывать и считывать данные с периферийного устройства, указывая определенные адреса, на которые оно реагирует. В некоторых микропроцессорах периферийные устройства фактически задействуют адреса, обычно используемые для обращения к памяти. Такая конфигурация называется *вводом-выводом с распределением памяти*. Тем не менее в процессоре 8080, кроме обычных 65 536 адресов для устройств ввода и вывода, специально зарезервированы 256 дополнительных, которые называются *портами ввода/вывода*. Адресные сигналы устройств ввода/вывода подаются на входы с A_0 по A_7, а от обращений к памяти их отличают сигналы, фиксируемые чипом системного контроллера 8228.

Команда OUT записывает содержимое аккумулятора в порт, адресуемый следующим за командой байтом. Команда IN позволяет считать байт в аккумулятор.

Код	Команда
D3	OUT pp
DB	IN pp

Периферийным устройствам иногда требуется привлечь внимание микропроцессора. Например, когда вы нажимаете клавишу на клавиатуре, желательно, чтобы микропроцессор узнавал об этом сразу. Это реализуется благодаря механизму *прерываний* — сигналам, поступающим от периферийного устройства на вход INT процессора 8080.

Однако после перезапуска микропроцессор 8080 не реагирует на прерывания. Для разрешения прерываний программа должна выполнить команду EI (Enable Interrupts — разрешить прерывания), а для их запрещения — команду DI (Disable Interrupts — запретить прерывания).

Код	Команда
F3	DI
FB	EI

Выходной сигнал процессора 8080 INTE означает, что прерывания были разрешены. Когда у периферийного устройства возникает необходимость прервать работу микропроцессора, оно подает на вход INT сигнал, равный 1. В ответ на него процессор 8080 извлекает из памяти команду, однако управляющие сигналы сообщают о прерывании. В ответ на это периферийное устройство передает процессору 8080 одну из следующих команд.

Код	Команда	Код	Команда
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
D7	RST 2	F7	RST 6
DF	RST 3	FF	RST 7

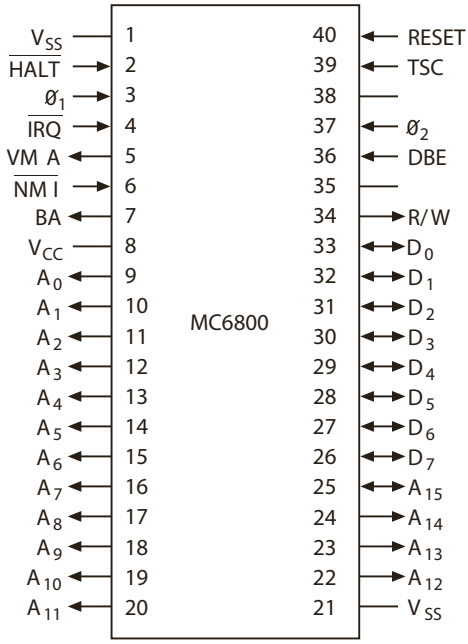
Эти команды RST (Restart — перезагрузка) аналогичны командам CALL в плане того, что текущее значение счетчика команд сохраняется в стеке. Однако после этого RST осуществляет переход к определенным адресам памяти: RST 0 переходит к ячейке 0000h, RST 1 — к ячейке 0008h и так далее вплоть до RST 7, которая совершает переход к ячейке 0038h. В этих ячейках должны содержаться фрагменты кода, предусмотренного для обработки прерывания. Например, прерывание от клавиатуры привело к выполнению команды RST 4. Это значит, что начиная с ячейки 0020h в памяти должен храниться некоторый код для считывания байта, введенного с клавиатуры. (В главе 21 я объясню все это.)

К настоящему моменту мной описаны 243 кода команд. Существует 12 байтов, которые не соответствуют никаким командам: 08h, 10h, 18h, 20h, 28h, 30h, 38h, CBh, D9h, DDh, EDh и FDh. В сумме это дает 255. Но есть еще один код, о котором я должен упомянуть.

Код	Команда
00	NOP

Команда NOP (No Operation — нет операции) заставляет процессор бездействовать. Для чего она нужна? Для заполнения адресного пространства. Как правило, процессор 8080 может выполнять множество команд NOP без каких-либо неприятных последствий.

Не буду подробно описывать устройство микросхемы 6800 компании Motorola, поскольку ее конструкция и функционал во многом аналогичны соответствующим аспектам процессора 8080. На следующей схеме изображены ее 40 контактов.



Контакт V_{SS} подключается к земле, V_{CC} — к источнику питания с напряжением пять вольт. Как и процессор 8080, микросхема 6800 предусматривает 16 выходных адресных сигналов и восемь сигналов для данных, работающих как на ввод, так и на вывод. Есть сигналы RESET и R/W (чтение/запись). На вход IRQ подается сигнал *запроса на прерывание* (interrupt request). Считается, что система синхронизации в микросхеме 6800 устроена гораздо проще, чем в процессоре 8080. Чего в микросхеме 6800 нет, так это понятия портов ввода/вывода. Адреса всех устройств ввода и вывода принадлежат общему адресному пространству 6800.

Микросхема 6800 предусматривает 16-битный счетчик команд, 16-битный указатель стека, 8-битный регистр состояния (для флагов) и два 8-битных аккумулятора, называемых A и B.

В качестве аккумулятора, а не простого регистра рассматривается B, поскольку его можно использовать так же, как и A. Однако дополнительных 8-битных регистров в такой микросхеме нет.

Вместо этого 6800 предполагает 16-битный *индексный регистр*, который может использоваться для хранения 16-битного адреса, подобно паре регистров HL в 8080. Для многих команд адрес может вычисляться путем суммирования значения их индексного регистра и байта, который следует за кодом команды.

Несмотря на то что микросхема 6800 выполняет примерно те же операции, что и процессор 8080 (загрузка, сохранение, сложение, вычитание, сдвиг,

переход, вызов), коды команд и соответствующие им мнемокоды у этих чипов совершенно разные. В следующей таблице, например, перечислены команды *перехода* из набора микросхемы 6800.

Код	Команда	Значение
20h	BRA	Переход
22h	BHI	Переход, если больше
23h	BLS	Переход, если меньше или равно
24h	BCC	Переход, если переноса нет
25h	BCS	Переход, если перенос есть
26h	BNE	Переход, если не равно
27h	BEQ	Переход, если равно
28h	BVC	Переход, если нет переполнения
29h	BVS	Переход, если есть переполнение
2Ah	BPL	Переход, если плюс
2Bh	BMI	Переход, если минус
2Ch	BGE	Переход, если больше или равно 0
2Dh	BLT	Переход, если меньше 0
2Eh	BGT	Переход, если больше 0
2Fh	BLE	Переход, если меньше или равно 0

В микросхеме 6800 не используется флаг четности, однако в отличие от 8080 в ней предусмотрен флаг переполнения. Некоторые из перечисленных команд перехода зависят от *комбинаций* флагов.

Разумеется, наборы команд 8080 и 6800 различаются. Эти два процессора были спроектированы примерно в одно и то же время двумя разными группами инженеров в двух разных компаниях. Эта несовместимость означает, что ни один из этих процессоров не может выполнить машинный код, написанный для другого. Кроме того, программа, написанная на языке ассемблера одного процессора, не может быть преобразована в коды команд другого. О написании компьютерных программ, работающих на нескольких процессорах, мы поговорим в главе 24.

Существует еще одно интересное различие между 8080 и 6800: в обоих микропроцессорах команда LDA загружает в аккумулятор значение из указанной ячейки памяти. В 8080, например, следующая последовательность байтов загружает в аккумулятор байт из ячейки 347Bh.

Теперь сравните эту последовательность с командой LDA для процессора 6800, использующей так называемый расширенный режим адресации.

Эта последовательность байтов загружает в аккумулятор A байт из ячейки 7B34h.

Различие не сразу бросается в глаза. Конечно, вы ожидали, что коды команд будут разными: 3Ah для 8080 и B6h для 6800. Однако эти два микропроцессора также по-разному обрабатывают адрес, который следует за кодом операции. Процессор 8080 предполагает, что первым должен идти младший байт, за которым следует старший, а процессор 6800 — что первым должен идти старший байт.

Это принципиальное различие между микросхемами Intel и Motorola по части хранения многобайтных значений *так и не было преодолено*. Микропроцессоры Intel по сей день продолжают сохранять многобайтные значения, начиная с младшего байта, а микропроцессоры Motorola — начиная со старшего байта.

Эти два порядка известны как «от младшего к старшему» (little-endian, способ Intel) и «от старшего к младшему» (big-endian, способ Motorola). Прежде чем спорить, какой из методов лучше, имейте в виду, что термины Big-Endian («тупоконечник») и Little-Endian («остроконечник») взяты из книги Джонатана Свифта «Путешествия Гулливера» и связаны с войной между Лилипутией и Блефуску, разгоревшейся из-за разногласий относительно того, с какого конца следует разбивать вареное яйцо: с острого или с тупого. Этот спор не имеет смысла. (С другой стороны, признаюсь, что подход, который я использовал в компьютере из главы 17, не кажется мне предпочтительным!) Хотя ни один из приведенных методов не может считаться правильным, разница между ними создает дополнительную проблему несовместимости при обмене информацией между системами, основанными на этих различающихся принципах.

Что же стало с двумя классическими микропроцессорами? Процессор 8080 был положен в основу того, что некоторые люди называли первым персональным компьютером, хотя его правильнее было бы назвать первым *домашним* компьютером. Это был «Альтаир 8800» (Altair 8800), фотография которого украсила обложку журнала Popular Electronics в январе 1975 года.

Если вы внимательно рассмотрите этот компьютер, то заметите на передней панели уже знакомые индикаторы и переключатели. Это тот же примитивный «пульт управления», который я предложил для массива RAM в главе 16.

Вслед за процессором 8080 были выпущены микросхемы Intel 8085 и Z-80 компании Zilog, конкурента Intel, основанного ее бывшим сотрудником Федерико Фаджином, который сделал существенный вклад в разработку микросхемы 4004. Процессор Z-80 был полностью совместим с 8080, но предусматривал множество дополнительных полезных команд. В 1977 году чип Z-80 был использован в микрокомпьютере компании Radio Shack TRS-80 Model 1.

Кроме того, в 1977 году компания Apple Computer Company, основанная Стивом Джобсом и Стивом Возняком, представила компьютер Apple II, в котором вместо 8080 и 6800 использовался малобюджетный чип 6502 компании MOS Technology — усовершенствованная версия микросхемы 6800.

В июне 1978 года компания Intel выпустила 16-битный микропроцессор 8086, способный адресовать один мегабайт памяти. Набор команд 8086 не был совместим с процессором 8080, однако он предусматривал команды для умножения и деления. Год спустя Intel представила микропроцессор 8088, внутренне идентичный чипу 8086, но с побайтной адресацией внешней памяти, позволявшей микропроцессору использовать распространенные в то время 8-битные вспомогательные чипы, разработанные для 8080. Компания IBM применяла микропроцессор 8088 в своем персональном компьютере 5150, представленном осенью 1981 года под названием IBM PC.

Выход IBM на рынок персональных компьютеров серьезно на него повлиял, и многие компании начали выпускать устройства, совместимые с IBM PC. (Тема совместимости будет рассмотрена в последующих главах.) На протяжении многих лет выражение «IBM PC-совместимый» подразумевало использование в устройстве микросхемы Intel, в частности микропроцессора Intel семейства x86. В 1982 году семейство x86 пополнилось чипами 186 и 286, в 1985 году — 32-битным чипом 386, в 1989-м — чипом 486, а в 1993-м — микропроцессорами Intel Pentium, которые в настоящее время устанавливаются в компьютеры, совместимые с IBM PC. Несмотря на то что наборы команд микропроцессоров Intel постоянно расширяются, они продолжают поддерживать коды команд более ранних процессоров, начиная с 8086.

В компьютере Apple Macintosh, впервые представленном в 1984 году, использовался 16-битный микропроцессор Motorola 68000, который является прямым потомком чипа 6800. Процессор 68000 и его более поздние версии (часто объединяемые в серию 68K) — один из самых популярных микропроцессоров.

Начиная с 1994 года в компьютерах Macintosh установлен микропроцессор PowerPC, разработанный совместно компаниями Motorola, IBM и Apple. В его основе лежит микропроцессорная архитектура RISC (Reduced Instruction Set Computing — вычисления с сокращенным набором команд), в рамках которой реализуется попытка увеличения скорости процессора за счет упрощения некоторых его аспектов. На компьютере с архитектурой RISC каждая команда, как правило, имеет одинаковую длину (32 бита в случае PowerPC), доступ к памяти ограничен только командами для загрузки и сохранения, а сами команды выполняют скорее простые операции, нежели сложные. Процессоры на основе архитектуры RISC обычно предусматривают множество регистров, чтобы как можно реже обращаться к памяти.

Микропроцессор PowerPC не может выполнять код, написанный для чипов серии 68K, поскольку имеет совершенно другой набор команд. Однако микропроцессоры PowerPC, которые в настоящее время используются в компьютерах Apple Macintosh, могут *эмулировать* процессор 68K. Программа-эмулятор, работающая на PowerPC, последовательно анализирует каждый код команды в программе, написанной для чипа серии 68K, и выполняет соответствующее действие. Это происходит не так быстро, как выполнение «родного» кода PowerPC, но это работает.

Согласно закону Мура, количество транзисторов в микропроцессорах должно удваиваться каждые 18 месяцев. Для чего нужны эти многочисленные дополнительные транзисторы?

Некоторые из транзисторов позволили увеличить разрядность процессора. Использование других было обусловлено появлением новых команд. Бóльшая

часть современных микропроцессоров предусматривает команды для выполнения операций над числами с плавающей точкой (о чем я расскажу в главе 23). Кроме того, в набор микропроцессоров были добавлены новые команды для произведения некоторых часто повторяющихся вычислений, необходимых для отображения на экранах компьютеров изображений или фильмов.

Для увеличения быстродействия в современных процессорах применяется несколько методов. Один из них называется *конвейеризацией*. При выполнении одной команды процессор считывает следующую, при этом он до определенной степени предугадывает, как команды перехода могут изменить программный поток. Современные процессоры также включают *кеш* (cache) — массив сверхскоростной оперативной памяти внутри процессора, в котором хранятся недавно выполненные команды. В компьютерных программах часто используются небольшие циклы, а кеш позволяет обойтись без повторной загрузки входящих в него команд. Все эти функции, повышающие быстродействие процессора, требуют дополнительных логических схем, следовательно, дополнительных транзисторов.

Как я уже говорил, микропроцессор — это только одна (пусть и самая важная) часть компьютерной системы. Сконструируем такую систему в главе 21, но сначала мы должны научиться записывать в память что-то, кроме кодов команд и чисел. Нам предстоит вернуться в первый класс и заново научиться читать и писать текст.

Глава 20
Набор символов ASCII

В цифровой памяти компьютера хранятся только биты, поэтому все, с чем мы собираемся работать, должно быть в виде битов. Мы уже видели, как с помощью битов можно представить числа и машинный код. Следующая задача — представить текст. В конце концов, большая часть накопленной в мире информации выражена в виде текста, а наши библиотеки полны книг, журналов и газет. Наверняка нам захочется использовать компьютеры для хранения звуков, изображений и фильмов, а с текста гораздо проще начать.

Чтобы представить текст в цифровой форме, мы должны разработать некоторую систему, в которой каждая буква соответствует уникальному коду. Для чисел и знаков препинания также нужно предусмотреть коды. Короче, нам нужны коды для всех *буквенно-цифровых* символов. Такая система иногда называется *набором кодированных символов*, а отдельные коды — *кодами символов*.

Сначала сформулируем вопрос: «Сколько битов требуется для этих кодов?» Ответить на него непросто!

Когда мы думаем о представлении текста с помощью битов, давайте не будем забегать далеко вперед. Мы привыкли к красиво отформатированному тексту на страницах книги, журнала или газеты. Абзацы состоят из строк одинаковой ширины. Однако такое форматирование существенно не влияет на сам текст. Когда мы читаем короткий рассказ в журнале, а спустя годы встречаем его в книге, он не кажется нам другим только потому, что в книге столбец текста шире.

Другими словами, забудьте, что текст отформатирован в виде плоских столбцов на печатной странице. Считайте его одномерным потоком букв, цифр, знаков препинания с дополнительным символом, обозначающим конец одного абзаца и начало следующего.

Опять же, если вы читаете рассказ в журнале, а позднее видите его в книге, и при этом шрифт немного отличается, имеет ли это какое-либо значение? Если журнальная версия начинается так:

Зовите меня Ишмаэль..., а книжная — так: Зовите меня Ишмаэль...

стоит ли обращать на это внимание? Вероятно, нет. Да, шрифт слегка влияет на восприятие, однако от его замены рассказ не теряет смысла. Исходный шрифт всегда можно вернуть. Это не наносит вреда.

Вот еще один способ упростить задачу: давайте использовать простой текст без курсива, полужирного начертания, подчеркивания, цветов, обводки букв, подстрочных и надстрочных индексов, а также без диакритических значков. Никаких Å, é, ñ или ö. Только символы латинского алфавита, поскольку из них состоит 99% английских слов.

В ходе предыдущих исследований кодов Морзе и Брайля мы видели, как буквы алфавита могут быть представлены в двоичном формате. Несмотря на то что эти системы прекрасно справляются со стоящими перед ними задачами, они имеют недостатки, когда речь заходит о компьютерах. В азбуке Морзе коды имеют *разную длину*. Например, часто используемые буквы обозначаются короткими кодами, более редкие — длинными. Как видите, такой код подходит для телеграфа, однако не для компьютеров. Кроме того, код Морзе не делает различий между буквами в верхнем и нижнем регистре.

Коды Брайля имеют фиксированную длину, каждая буква представлена шестью битами, что предпочтительнее для компьютеров. Азбука Брайля также различает букву верхнего и нижнего регистра, хотя и использует для этого специальный *escape*-код, указывающий, что за ним следует символ в верхнем регистре. По сути, это означает, что для каждой заглавной буквы требуются два кода, а не один. Цифры отображаются с помощью кода *переключения*, который дает понять, что следующие далее коды представляют числа до тех пор, пока не встретится другой код переключения, сигнализирующий возврат к представлению букв.

Наша цель — разработка такого набора кодированных символов, чтобы нижеприведенное предложение можно было зашифровать с помощью серии кодов, каждый из которых — это определенное количество битов.

У меня 27 сестер.

Одни коды будут представлять буквы, другие — знаки препинания, третьи — числа. Еще нам необходим код, соответствующий пробелу между словами. Приведенное предложение состоит из 17 символов (включая пробелы). Последовательность кодов для шифрования подобных предложений часто называется текстовой *строкой*.

То, что нам нужны коды для чисел в текстовой строке, таких как «27», может показаться странным, поскольку мы представляли числа с помощью битов во многих предыдущих главах. Мы можем предположить, что кодами для

цифр 2 и 7 в этом предложении являются просто двоичные числа 10 и 111. Однако это не обязательно так. В контексте такого предложения с символами 2 и 7 можно обращаться как с любыми другими символами в письменном языке. Им могут соответствовать коды, совершенно не связанные с фактическими значениями этих чисел.

Вероятно, наиболее экономичным текстовым кодом является 5-битный код, созданный в 1874 году для печатающего телеграфа Эмилем Бодо*, сотрудником французской телеграфной службы, которая начала использовать этот код в 1877 году. В дальнейшем код был усовершенствован Дональдом Мюрреем и стандартизирован в 1931 году Международным консультационным комитетом по телефонии и телеграфии (Comité Consultatif International Téléphonique et Télégraphique, CCITT; сейчас — Международный союз электросвязи — International Telecommunication Union, ITU). Официально этот код называется международным телеграфным алфавитом № 2 (International Telegraph Alphabet No 2, ITA-2); в Соединенных Штатах он более известен как код Бодо, хотя правильнее было бы называть его кодом Мюррея.

В XX веке код Бодо часто применялся в *телетайпных аппаратах*. Клавиатура телетайпа Бодо похожа на пишущую машинку, но у нее только 30 клавиш и пробел. Клавиши телетайпа — просто переключатели, использование которых приводит к генерации двоичного кода и его передаче по выходному кабелю аппарата, бит за битом. Кроме того, телетайп предусматривает печатный механизм. Коды, проходящие через входной кабель телетайпа, активируют электромагниты, которые печатают символы на бумаге.

Поскольку код Бодо 5-битный, он содержит всего 32 элемента. Шестнадцатеричные значения этих кодов находятся в диапазоне от 00h до 1Fh. В следующей таблице представлено соответствие этих 32 кодов буквам латинского алфавита.

Шестнадцатеричный код	Буква Бодо	Шестнадцатеричный код	Буква Бодо
00		10	E
01	T	11	Z
02	*Возврат каретки*	12	D
03	O	13	B
04	*Пробел*	14	S
05	H	15	Y
06	N	16	F

* В честь Эмиля Бодо символьная скорость измеряется в бодах. *Прим. науч. ред.*

Шестнадцатеричный код	Буква Бодо	Шестнадцатеричный код	Буква Бодо
07	M	17	X
08	*Перевод строки*	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	*Переключение на цифры*
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K
0F	V	1F	*Переключение на буквы*

Код 00h не присваивается ничему. Из оставшихся кодов двадцать шесть назначаются буквам алфавита, а остальные пять соответствуют вспомогательным действиям, выделенным в таблице курсивом.

Код 04h — это пробел, который создает пространство между словами, коды 02h и 08h — возврат каретки и перевод строки. Эти понятия применяются при использовании пишущей машинки. Когда достигаете конца строки, печатая, вы нажимаете на рычаг или кнопку, которая выполняет два действия. Во-первых, каретка перемещается вправо, благодаря чему следующая строка начинается с левого края листа (возврат каретки). Во-вторых, пишущая машинка прокручивает валик так, чтобы следующая строка находилась под той, которую вы только что напечатали (перевод строки). В системе Бодо эти два кода генерируются отдельными клавишами. Телетайпный аппарат Бодо реагирует на эти два кода при печати.

Для получения цифр и знаков препинания в системе Бодо используется код 1Bh, обозначенный в таблице фразой «Переключение на цифры». Все следующие за ним коды интерпретируются как цифры или знаки препинания, пока код «Переключение на буквы» (1Fh) не просигнализирует возврат к буквам. В следующей таблице представлены коды, соответствующие цифрам и знакам препинания.

Шестнадцатеричный код	Символ Бодо	Шестнадцатеричный код	Символ Бодо
00		10	3
01	5	11	+
02	*Возврат каретки*	12	*Кто это?*
03	9	13	?
04	*Пробел*	14	‘
05	#	15	6
06	,	16	$
07	.	17	/
08	*Перевод строки*	18	—
09	)	19	2
0A	4	1A	*Сигнал*
0B	&	1B	*Переключение на цифры*
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	*Переключение на буквы*

В стандарте ITU коды 05h, 0Bh и 16h не определены: они зарезервированы «для национального использования». В таблице показано, как эти коды применялись в Соединенных Штатах. Эти же коды обычно подходят для букв с диакритическими знаками из некоторых европейских языков. Код «Сигнал» предназначен для подачи телетайпом слышимого звукового сигнала, «Кто это?» активирует механизм, посредством которого телетайп может идентифицировать себя.

Как и азбука Морзе, этот 5-битный код не предусматривает различий между прописными и строчными буквами. Предложение I spent $25 today («Сегодня я потратил 25 долларов») шифруется следующей последовательностью шестнадцатеричных значений.

I SPENT $25 TODAY.

0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08

Обратите внимание на три кода переключения: 1Bh прямо перед числом, 1Fh после числа и 1Bh перед точкой в конце предложения. Строка завершается кодами возврата каретки и перевода строки.

К сожалению, если вы дважды отправите эту последовательность значений на печатающее устройство телетайпа, получите следующий результат.

I SPENT $25 TODAY.
8 '03,5 $25 TODAY.

Что случилось? Дело в том, что последний код переключения, полученный печатным аппаратом перед второй строкой, представлял код переключения на цифры, поэтому коды в начале второй строки были интерпретированы как цифры.

Подобные проблемы типичны при использовании кодов переключения. Несмотря на безусловную экономичность кода Бодо, предпочтительнее было бы остановиться на уникальных кодах для цифр и знаков препинания, а также отдельных кодах для строчных и прописных букв.

Если мы хотим выяснить, сколько бит нам необходимо для создания лучшей по сравнению с кодом Бодо системы кодирования символов, нужно просто сложить их: 52 кодовых слова — для прописных и строчных букв, десять кодовых слов — для цифр от 0 до 9. Это уже 62 кодовых слова. Если добавить несколько знаков препинания, получим 64 кодовых слова, а значит, нам нужно более шести бит. Однако мы еще не скоро превысим значение в 128 символов, при котором потребуются восемь бит.

Как видите, необходимы семь бит для представления символов английского текста, если мы хотим обойтись без переключения между буквами нижнего и верхнего регистра.

Что же это за коды? Фактически коды могут быть любыми. Если бы мы собирались создать собственный компьютер, собрать для этого все необходимые аппаратные средства, сами его запрограммировать и никогда не использовать для подключения к любой другой машине, мы могли бы придумать собственные коды. Все, что нужно, — это назначить уникальный код каждому символу, который мы собираемся использовать.

Поскольку компьютеры редко создаются и работают отдельно друг от друга, разумнее было бы договориться о применении одних и тех же кодов. Таким образом сконструированные нами компьютеры могут быть более совместимыми друг с другом и, вероятно, смогут даже обмениваться текстовой информацией.

Кроме того, не следует присваивать коды случайным образом. Так, когда мы работаем с текстом на компьютере, назначение последовательных кодов буквам алфавита дает некоторые преимущества. Например, этот метод упорядочивания упрощает сортировку по алфавиту.

К счастью, такой стандарт уже разработан: *Американский стандартный код для обмена информацией* (American Standard Code for Information Interchange, ASCII). Он был принят в 1967 году и остается одним из важнейших во всей компьютерной индустрии. Однако у него есть одно исключение (о котором расскажу позднее). Работая с текстом на компьютере, вы можете быть уверены, что стандарт ASCII каким-то образом задействован в этом процессе.

ASCII — это 7-битный код, использующий двоичные значения в диапазоне от 0000000 до 1111111, которые соответствуют шестнадцатеричным кодам от 00h до 7Fh. Давайте разберем коды ASCII. Начнем не с самого начала, поскольку первые 32 кода концептуально немного сложнее остальных, а со второй группы, состоящей из 32 кодов, которая включает знаки препинания и десять цифр. В следующей таблице перечислены шестнадцатеричные коды и соответствующие им символы.

Шестнадцатеричный код	**Символ ASCII**	**Шестнадцатеричный код**	**Символ ASCII**
20	*Пробел*	30	0
21	!	31	1
22	«	32	2
23	#	33	3
24	$	34	4
25	%	35	5
26	&	36	6
27	‘	37	7
28	(	38	8
29	)	39	9
2A	*	3A	:
2B	+	3B	;
2C	,	3C	<
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

Обратите внимание: код 20h соответствует пробелу, отделяющему слова друг от друга.

Следующие 32 кода — прописные буквы и некоторые дополнительные знаки препинания. Кроме значка @ и символа подчеркивания, эти знаки обычно отсутствуют на пишущих машинках, однако являются стандартными для компьютерных клавиатур.

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
40	@	50	P
41	A	51	Q
42	B	52	R
45	C	55	S
44	D	54	T
45	E	55	U
46	F	56	V
47	G	57	W
48	H	58	X
49	I	59	Y
4A	J	5A	Z
4B	K	5B	[
4C	L	5C	\
4D	M	5D	]
4E	N	5E	^
4F	O	5F	_

Следующая группа из 32 символов: все строчные буквы и некоторые дополнительные знаки препинания, которые, опять же, редко встречаются на пишущих машинах.

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
60	‘	70	p
61	a	71	q
62	b	72	r
63	c	73	s
64	d	74	t
65	e	75	u
66	f	76	v
67	g	77	w
68	h	78	x
69	i	79	y
6A	j	7A	z
6B	k	7B	{
6C	l	7C	|
6D	m	7D	}
6E	n	7E	~
6F	o		

В этой таблице отсутствует последний символ, соответствующий коду 7Fh. Приведенные выше три таблицы содержат в общей сложности 95 символов. Поскольку код ASCII является 7-битным, он допускает использование 128 кодов, поэтому нам должно быть доступно еще 33 кода, до которых мы скоро и доберемся.

Текстовая строка Hello, you! («Привет тебе!») может быть представлена в кодировке ASCII с использованием следующих шестнадцатеричных кодов.

Hello, you!

48 65 6C 6C 6F 2C 20 79 6F 75 21

Обратите внимание на запятую (код 2C), пробел (код 20) и восклицательный знак (код 21), а также на коды, соответствующие буквам. Представление короткого предложения I am 12 years old («Мне 12 лет») в кодировке ASCII следующее.

I am 12 years old.

49 20 61 6D 20 31 32 20 79 65 61 72 73 20 6F 6C 64 2E

Число 12 в этом предложении отображено шестнадцатеричными числами 31h и 32h, то есть кодами ASCII для цифр 1 и 2. Когда число 12 — часть текста, его *нельзя* представлять с помощью шестнадцатеричных кодов 01h и 02h, BCD-кода 12h или шестнадцатеричного кода 0Ch. Все эти коды имеют в системе ASCII совершенно другие значения.

Конкретная прописная буква в системе ASCII отличается от своей строчной версии на 20h. Этот факт позволяет легко написать код, который, например, переводит текстовую строку в верхний регистр. Предположим, что определенная область памяти содержит текстовую строку, по одному символу на байт. Следующая подпрограмма для процессора 8080 предполагает, что адрес первого символа в текстовой строке хранится в регистре HL. Регистр C включает длину этой текстовой строки, то есть количество символов.

```
Capitalize:    MOV A, C      ; C = количество оставшихся символов
               CPI A,00h     ; Сравнить с 0
               JZ AllDone    ; Если C = 0, завершить программу
```

```
                MOV A,[HL]    ; Извлечь следующий символ
                CPI A,61h     ; Меньше, чем «a»?
                JC SkipIt     ; Если да, проигнорировать
                CPI A,7Bh     ; Больше, чем «z»?
                JNC SkipIt    ; Если да, проигнорировать

                SBI A,20h     ; Буква в нижнем регистре, поэтому
                              ; вычитаем 20h
                MOV [HL], A   ; Сохранить символ
SkipIt:         INX HL ; Перейти к следующему символу
                DCR C         ; Уменьшить счетчик на 1
                JMP Capitalize; Вернуться к началу

AllDone:        RET
```

Оператор, который вычитает 20h из кода строчной буквы для ее преобразования в прописную, можно заменить следующим фрагментом.

```
ANI A, DFh
```

Инструкция ANI (AND Immediate) выполняет побитовую операцию И между значением в аккумуляторе и значением DFh, которое в двоичном формате равно 11011111. Под словом «побитовая» подразумеваю, что такая команда выполняет операцию И между каждой парой соответствующих битов, составляющих два числа. Эта операция И сохраняет все биты, содержащиеся в аккумуляторе (A), за исключением третьего слева, значение которого задается равным 0, что, в свою очередь, преобразует строчную букву ASCII в прописную.

Приведенные выше 95 кодов — это так называемые *печатные символы*, поскольку они имеют графическое представление. Кроме них в кодировке ASCII также предусмотрено 33 *управляющих символа*, которые не имеют графического представления, но выполняют определенные функции. Для полноты приведу эти 33 управляющих символа. Не беспокойтесь, если они покажутся вам непонятными. Кодировка ASCII изначально предназначалась для использования в телетайпах, поэтому многие из ее кодов в настоящее время потеряли свое значение.

Шестнадцатеричный код	Аббревиатура	Название управляющего символа	Назначение
00	NUL	Null	Нет
01	SOH	Start of Heading	Начало заголовка
02	STX	Start of Text	Начало текста
03	ETX	End of Text	Конец текста
04	EOT	End of Transmission	Конец передачи
05	ENQ	Enquiry	Запрос
06	ACK	Acknowledge	Подтверждение
07	BEL	Bell	Звуковой сигнал
08	BS	Backspace	Возврат на символ назад
09	HT	Horizontal Tabulation	Горизонтальная табуляция
0A	LF	Line Feed	Перевод строки
0B	VT	Vertical Tabulation	Вертикальная табуляция
0C	FF	Form Feed	Новая страница
0D	CR	Carriage Return	Возврат каретки
0E	SO	Shift-Out	Переключиться на другую кодировку
0F	SI	Shift-In	Переключиться на исходную кодировку
10	DLE	Data Link Escape	Освобождение канала данных
11	DC1	Device Control 1	Первый символ управления устройством
12	DC2	Device Control 2	Второй символ управления устройством
13	DC3	Device Control 3	Третий символ управления устройством
14	DC4	Device Control 4	Четвертый символ управления устройством
15	NAK	Negative Acknowledge	Отсутствие подтверждения

Шестнадцатеричный код	Аббревиатура	Название управляющего символа	Назначение
16	SYN	Synchronous Idle	Синхронизация
17	ETB	End of Transmission Block	Конец блока передачи
18	CAN	Cancel	Отмена
19	EM	End of Medium	Конец носителя
1A	SUB	Substitute Character	Замена
1B	ESC	Escape	Ключ
1C	FS	File Separator or Information Separator 4	Разделитель файлов или информации 4
1D	GS	Group Separator or Information Separator 3	Разделитель групп или информации 3
1E	RS	Record Separator or Information Separator 2	Разделитель записей или информации 2
1F	US	Unit Separator or Information Separator 1	Разделитель элементов или информации 1
7F	DEL	Delete	Удаление

Управляющие символы в данном случае могут использоваться наряду с печатными для элементарного форматирования текста. Это проще всего понять, если вы подумаете о таком устройстве, как телетайп или простой принтер, который отпечатывает на странице символы под воздействием потока кодов ASCII. Получив код символа, печатающая головка устройства наносит символ и перемещается на одну позицию вправо. Наиболее важные управляющие символы изменяют это нормальное поведение.

Рассмотрим следующую строку шестнадцатеричных значений.

```
41 09 42 09 43 09
```

Символ 09 — это код горизонтальной табуляции, или табулятор. Если представить, что все горизонтальные позиции символа на странице принтера нумеруются начиная с 0, то код табуляции дает команду напечатать следующий символ на следующей горизонтальной позиции, номер которой кратен восьми, например так.

```
A       B       C
```

Это удобный способ для расположения текста столбцами.

Даже сегодня многие компьютерные принтеры реагируют на код перехода к новой странице (12h), извлекая текущую страницу и начиная печатать на новой.

Код для возврата на один символ назад (backspace) может использоваться для печати составных символов на некоторых старых принтерах. Предположим, вам нужно, чтобы телетайпный аппарат отобразил строчную букву *e* с обратным апострофом: *è*. Эту задачу можно решить с помощью шестнадцатеричных кодов 65 08 60.

Сегодня наиболее важными являются управляющие коды для возврата каретки и перевода строки, которые имеют то же значение, что и аналогичные коды Бодо. В ответ на код возврата каретки печатающая головка принтера перемещается к левому краю страницы, а в ответ на код перевода строки — на одну строку вниз. Обычно для перехода на новую строку требуются оба кода. Код возврата каретки может использоваться сам по себе для печати поверх существующей строки, а код перевода строки — для того, чтобы перейти к следующей, при этом не перемещаясь к левому краю страницы.

Несмотря на то что кодировка ASCII — доминирующий стандарт в компьютерном мире, она не используется во многих крупных компьютерных системах IBM. Для мейнфреймов System/360 компания разработала собственный 8-битный код EBCDIC (Extended BCD Interchange Code — расширенный двоично-десятичный код обмена информацией) — расширенный вариант более раннего 6-битного кода BCDIC, полученного из кодов, используемых на перфокартах IBM. Эти перфокарты, позволяющие хранить по 80 текстовых символов, были созданы IBM в 1928 году и использовались на протяжении более 50 лет.

Рассматривая взаимосвязь между перфокартами и соответствующими им 8-битными кодами символов EBCDIC, имейте в виду, что эти коды разрабатывались на протяжении многих десятилетий под влиянием различных

технологий. По этой причине не стоит ожидать от них излишней логичности или согласованности.

На перфокарте символ кодируется комбинацией из одного или нескольких прямоугольных отверстий, пробитых в одном столбце. Сам символ часто печатается в верхней части карты. Нижние десять строк пронумерованы от 0 до 9. Ненумерованная строка над нулевой строкой считается одиннадцатой, а верхняя — двенадцатой; десятая строка отсутствует.

Вот еще несколько терминов из области применения перфокарт IBM: строки с нулевой по девятую называются *цифровыми строками*, или *цифровой пробивкой*. Одиннадцатая и двенадцатая строки — *зонные строки*, или *зонная пробивка*. Иногда нулевая и девятая строки считались не цифровыми, а зонными, что вызывало путаницу.

Восьмибитный код символа EBCDIC состоит из старшей и младшей тетрад (четыре бита). Младшая тетрада — код BCD, соответствующий цифровой пробивке символа; старшая тетрада — код, который произвольно можно поставить в соответствие зонной пробивке символа. Из главы 19 вы помните, что BCD означает *двоично-десятичный код* — 4-битный код для цифр от 0 до 9.

Для цифр от 0 до 9 не существует никакой зонной пробивки. Отсутствие пробивки соответствует старшей тетраде 1111. Младшая тетрада — код BCD цифровой пробивки. В следующей таблице приведены коды EBCDIC для цифр от 0 до 9.

Шестнадцатеричный код	Символ EBCDIC
F0	0
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9

Для прописных букв тетрада 1100 соответствует зонной пробивке только двенадцатой строки, тетрада 1101 — зонной пробивке только одиннадцатой строки, тетрада 1110 — зонной пробивке только нулевой строки. Приведем коды EBCDIC для прописных букв.

Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC
C1	A	D1	J		
C2	B	D2	K	E2	S
C3	C	D3	L	E3	T
C4	D	D4	M	E4	U
C5	E	D5	N	E5	V
C6	F	D6	O	E6	W
C7	G	D7	P	E7	X
C8	H	D8	Q	E8	Y
C9	I	D9	R	E9	Z

Обратите внимание на зазоры в нумерации этих кодов. Если вы используете текст EBCDIC при написании программ, эти зазоры могут мешать.

Строчные буквы соответствуют той же цифровой пробивке, что и прописные, но другой зонной пробивке. Для строчных букв от *a* до *i* пробиваются двенадцатая и нулевая строки, что соответствует коду 1000, для букв от *j* до *r* — двенадцатая и одиннадцатая строки, что соответствует коду 1001, для букв от *s* до *z* — одиннадцатая и нулевая строки, что соответствует коду 1010. Коды EBCDIC для строчных букв следующие.

Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC
81	a	91	j		
82	b	92	k	A2	s
83	c	93	l	A3	t
84	d	94	m	A4	u
85	e	95	n	A5	v
86	f	96	o	A6	w
87	g	97	p	A7	x
88	h	98	q	A8	y
89	i	99	r	A9	z

Разумеется, существуют и другие коды EBCDIC — для знаков препинания и управляющих символов, однако мы едва ли нуждаемся в проведении полномасштабного исследования этой системы.

На первый взгляд может показаться, что одного столбца перфокарты IBM достаточно для кодирования 12 бит информации. Каждое отверстие соответствует одному биту, не так ли? По идее, для кодирования символа ASCII должно быть достаточно семи из 12 позиций в каждом столбце. Однако на практике

это не очень хорошо работает, поскольку при этом пробивается слишком много отверстий, из-за чего карта становится хрупкой.

Многие из 8-битных кодов EBCDIC не определены. Это говорит о том, что использование 7-битной кодировки ASCII имеет больше смысла. Во времена разработки системы ASCII память была дорогостоящей. Некоторые люди полагали, что кодировка ASCII должна быть 6-битной и предусматривать символ переключения между строчными и прописными буквами для экономии.

Как только эта идея была отвергнута, другие стали полагать, что кодировка ASCII должна быть 8-битной, поскольку даже в то время считалось, что в компьютерах будет применяться скорее 8-битная архитектура, чем 7-битная. Конечно, современным стандартом являются 8-битные байты. Несмотря на то что технически ASCII — это 7-битная кодировка, почти всегда ее коды хранятся как 8-битные значения.

Эквивалентность байтов и символов, безусловно, удобна, поскольку мы можем приблизительно представить, какой объем компьютерной памяти занимает конкретный текстовый документ, просто подсчитав количество символов. Некоторым людям гораздо легче понять, что такое килобайт и мегабайт памяти, когда этот объем ставится в соответствие объему текста.

Например, обычная машинописная страница формата A4 с полями 2,5 сантиметра и двойным междустрочным интервалом содержит примерно 27 строк текста. На каждой строке шириной 16 сантиметров содержится 65 символов. Содержимое такой страницы занимает в общей сложности около 1750 байт. Текст, содержащийся на машинописной странице с одинарным междустрочным интервалом, занимает примерно вдвое больше — 3,5 килобайта.

Страница в журнале New Yorker включает три столбца текста, в каждом из которых содержатся 60 строк по 40 символов. Это 7200 символов (байтов) на страницу.

Страница газеты New York Times содержит шесть столбцов текста. Если бы вся она была занята текстом без заголовков или изображений (что было бы необычно), то каждый столбец состоял бы из 155 строк по 35 символов. Тогда на всей странице было бы 32 550 символов, или 32 килобайта.

На странице обычной книги насчитывается около 500 слов. В среднем слово состоит примерно из семи букв, хотя скорее из восьми, если учитывать пробел. Таким образом, на странице книги около 3000 символов. Предположим, что средняя книга состоит из 333 страниц. Это значение, каким бы странным оно ни казалось, позволяет сказать, что объем текста средней книги составляет около одного миллиона байт, или один мегабайт.

Разумеется, объем текста книг варьируется в большом диапазоне:

— «Великий Гэтсби» Фрэнсиса Скотта Фицджеральда — около 300 килобайт;
— «Над пропастью во ржи» Джерома Сэлинджера — около 400 килобайт;
— «Приключения Гекльберри Финна» Марка Твена — около 540 килобайт;
— «Гроздья гнева» Джона Стейнбека — около одного мегабайта;
— «Моби Дик, или Белый кит» Германа Мелвилла — 1,3 мегабайта;
— «История Тома Джонса, найденыша» Генри Филдинга — 2,25 мегабайта;
— «Унесенные ветром» Маргарет Митчелл — 2,5 мегабайта;
— «Противостояние» Стивена Кинга — 2,7 мегабайта;
— «Война и мир» Льва Толстого — 3,9 мегабайта;
— «В поисках утраченного времени» Марселя Пруста — 7,7 мегабайта.

В Библиотеке Конгресса Соединенных Штатов насчитывается около 20 миллионов книг, в которых содержится в общей сложности 20 триллионов символов, что соответствует 20 терабайтам текстовых данных. (Кроме текста, там находится множество фотографий и аудиозаписей.)

Несмотря на то что ASCII, безусловно, является основным стандартом в компьютерной индустрии, он не идеален. Проблема в том, что этот стандарт слишком американский! Действительно, ASCII не вполне подходит даже для тех стран, в которых основным языком является английский. Кодировка ASCII включает символ доллара, но где же символ британского фунта? А как насчет букв с диакритическими значками, используемыми во многих западноевропейских языках? Я уже не говорю о нелатинских алфавитах, таких как греческий, арабский, иврит и кириллица. А что насчет символов слогового письма брахми, применяемого в Индии и Юго-Восточной Азии, на котором основаны такие виды письменности, как деванагари, бенгали, тайская и тибетская? Как с помощью 7-битного кода в принципе можно представить *десятки тысяч* идеограмм китайского, японского и корейского языков, а также десять с лишним тысяч хангыльских слогов?

В период разработки системы ASCII потребностям некоторых других стран уделяли внимание, хотя нелатинские алфавиты при этом особо не учитывались. Согласно опубликованному стандарту ASCII, десять его кодов (40h, 5Bh, 5Ch, 5Dh, 5Eh, 60h, 7Bh, 7Ch, 7Dh и 7Eh) можно переопределить в соответствии с национальными потребностями. Кроме того, при необходимости символ решетки (#) можно заменить символом британского фунта (£), а символ доллара ($) — обобщенным для валюты символом (¤). Очевидно, что замена символов имеет смысл только тогда, когда все пользователи конкретного текстового документа, содержащего эти переопределенные коды, знают об этом изменении.

Поскольку многие компьютерные системы хранят символы в виде 8-битных значений, можно расширить их набор со 128 до 256. В таком наборе коды с 00h по 7Fh определяются так же, как и в обычной системе ASCII, а коды с 80h по FFh могут представлять что-то совершенно иное. Этот метод использовался для определения дополнительных кодов для букв с диакритическими значками и нелатинских алфавитов. В качестве примера приведу набор кодов для букв кириллицы. В представленной таблице старшая тетрада шестнадцатеричного кода символа указана в верхней строке, а младшая — в левом столбце.

	8-	**9-**	**A-**	**E-**
-0	А	Р	а	р
-1	Б	С	б	с
-2	В	Т	в	т
-3	Г	У	г	у
-4	Д	Ф	д	ф
-5	Е	Х	е	х
-6	Ж	Ц	ж	ц
-7	З	Ч	з	ч
-8	И	Ш	и	ш
-9	Й	Щ	й	щ
-A	К	Ъ	к	ъ
-B	Л	Ы	л	ы
-C	М	Ь	м	ь
-D	Н	Э	н	э
-E	О	Ю	о	ю
-F	П	Я	п	я

Символом кода A0h назначен неразрывный пробел. Обычно когда компьютерная программа форматирует текст в строки и абзацы, то разрыв строки равен пробелу, код ASCII которого 20h. Код A0h должен отображаться как пробел, но не может использоваться для разрыва строк. Неразрывный пробел может понадобиться, например, в фразе «WW II». Символ кода ADh — мягкий перенос. Его используют для разделения гласных в середине слова. На печатаемой странице он появляется, только когда необходимо перенести слово с одной строки на другую.

К сожалению, за минувшие десятилетия было создано много *разных* расширений кодировки ASCII, что привело к большой путанице и негативно отразилось на совместимости. Набор ASCII был расширен более радикальным образом для кодирования идеограмм китайского, японского и корейского языков. В одной популярной кодировке под названием Shift-JIS (Japan Industrial Standard — японский промышленный стандарт) коды с 81h по 9Fh фактически представляют первый байт двухбайтового кода символа. Таким образом, система Shift-JIS позволяет кодировать около 6000 дополнительных символов. К сожалению, Shift-JIS — не единственная система, которая использует такой подход. В Азии широко распространены еще три стандартных набора двухбайтовых символов (Double-byte character sets, DBCS).

Существование нескольких несовместимых наборов двухбайтовых символов — лишь одна из проблем. Печально, что некоторые символы, в частности обычные символы ASCII, представлены однобайтовыми кодами, а тысячи идеограмм — 2-байтовыми. Это затрудняет работу с такими наборами.

Считая предпочтительным наличие единой однозначной системы кодирования символов, подходящей для всех языков мира, в 1988 году несколько крупных компьютерных компаний объединились для разработки альтернативы ASCII, получившей название Unicode. В отличие от 7-битного кода ASCII, Unicode — 16-битный. Для каждого символа в кодировке Unicode требуется два байта. Значит, Unicode предусматривает коды символов от 0000h до FFFFh, то есть может представлять 65 536 различных символов. Этого достаточно для охвата всех языков мира, которые с большой долей вероятности будут использоваться в компьютерной индустрии, даже с возможностью расширения.

Кодировка Unicode создавалась не с нуля. Первые 128 символов Unicode, коды которых находятся в диапазоне от 0000h до 007Fh, соответствуют тем же символам в системе ASCII. Кроме того, коды Unicode с 00A0h по 00FFh — это коды описанного выше расширения ASCII для латинского алфавита Latin Alphabet No 1. В Unicode также включены другие мировые стандарты.

Несмотря на то что Unicode — очевидное улучшение систем кодировки, это не гарантирует его мгновенного принятия. Система ASCII и множество несовершенных расширений настолько укоренились в мире компьютерных технологий, что вытеснить их будет сложно.

Единственная настоящая проблема системы Unicode в том, что она делает недействительным прежнее соответствие между одним текстовым символом и одним байтом памяти. Закодированный с помощью стандарта ASCII роман «Гроздья гнева» занимает один мегабайт, а в кодировке Unicode — два мегабайта. Однако это небольшая плата за универсальную и однозначную систему кодирования.

Глава 21
Шины

Несмотря на основополагающую роль процессора, им устройство компьютера не ограничивается. Помимо него компьютеру требуется оперативная память (RAM) для хранения машинного кода, который будет выполнять процессор. Кроме того, компьютер должен предусматривать способ записи этого кода в память (устройство ввода) и отображения результатов работы программы (устройство вывода). Как вы помните, память RAM является энергозависимой: ее содержимое теряется при отключении питания. Так что еще один полезный компонент компьютера — долговременное запоминающее устройство, в котором код и данные могут храниться после его выключения.

Все интегральные схемы (ИС), из которых состоит компьютер, монтируются на печатных платах. В некоторых небольших компьютерах все ИС могут поместиться на одной плате. Однако гораздо чаще различные компоненты размещаются на двух или более платах, которые обмениваются данными с помощью *шины*. Шина — это просто набор цифровых сигналов, передаваемых по различным средам, которые подаются на каждую из плат компьютера. Эти сигналы делятся на четыре категории:

— адресные сигналы — сигналы, генерируемые микропроцессором и используемые в основном для адресации оперативной памяти; они также применяются для обращения к другим подключенным к компьютеру устройствам;
— сигналы вывода данных — эти сигналы тоже генерируются микропроцессором и используются для записи данных в оперативную память или их передачи в другие устройства; будьте осторожны с терминами «ввод» и «вывод»: сигнал вывода данных микропроцессора — сигнал ввода данных для оперативной памяти и других устройств;
— сигналы ввода данных — сигналы, которые генерируются другими компонентами компьютера и считываются микропроцессором; сигналы ввода данных чаще всего возникают на выходах RAM, благодаря чему

микропроцессор считывает содержимое памяти, однако и другие компоненты также генерируют сигналы ввода данных для микропроцессора;

— управляющие сигналы — разнообразные сигналы, которые обычно соответствуют управляющим сигналам конкретного микропроцессора, на базе которого построен компьютер; управляющие сигналы могут генерироваться в микропроцессоре или в других устройствах, которым требуется передать данные в микропроцессор; пример управляющего сигнала — сигнал, с помощью которого микропроцессор указывает на необходимость записи некоторых данных в конкретную ячейку памяти.

В дополнение к этому шина подает питание на различные платы, входящие в состав компьютера.

Одной из первых популярных шин для ПК была модель S-100, представленная в 1975 году в качестве компонента первого персонального компьютера «Альтаир» компании MITS. Несмотря на то что эта шина разрабатывалась для микропроцессора 8080, позднее она была адаптирована под другие процессоры, такие как 6800. Размер платы S-100 составляет 13,4 × 25,4 сантиметра. Одна из сторон печатной платы вставляется в разъем, который имеет 100 контактов (отсюда и название).

Компьютер с шиной S-100 включает в себя большую плату, называемую *материнской*, которая содержит несколько (около двенадцати) связанных друг с другом гнезд для плат S-100. Эти гнезда иногда называются *слотами расширения*, в них вставляются платы S-100, или *платы расширения*. Одну плату S-100 занимает микропроцессор 8080 и вспомогательные чипы (о некоторых я упомянул в главе 19). Оперативная память занимает одну или несколько других плат.

Поскольку шина S-100 разрабатывалась для микросхемы 8080, она имеет 16 адресных линий, восемь линий для ввода и восемь линий для вывода данных. Как вы помните, в самом процессоре 8080 линии для ввода и вывода данных объединены. Сигналы разделяются на входные и выходные с помощью других микросхем, установленных на той же плате, что и процессор 8080. Шина также предусматривает восемь линий для *прерываний* сигналов, генерируемых другими устройствами, когда им требуется привлечь внимание центрального процессора. Как мы увидим далее, клавиатура может генерировать сигнал прерывания при нажатии клавиши. В ответ на это процессор 8080 выполняет короткую программу, чтобы определить, какая клавиша была нажата, и предпринять соответствующее действие. Для обработки прерываний к плате с процессором 8080 обычно также подключается чип Intel 8214 (устройство для управления приоритетными прерываниями). Когда возникает прерывание,

этот чип генерирует для процессора 8080 должный сигнал. Когда последний подтверждает получение запроса на прерывание, чип посылает команду RST (Restart — перезапуск), которая заставляет микропроцессор сохранить текущее значение счетчика команд и в зависимости от полученного прерывания перейти к команде в ячейке 0000h, 0008h, 0010h, 0018h, 0020h, 0028h, 0030h или 0038h.

Если бы вы разработали новую компьютерную систему с шиной нового типа, нужно было бы решить, опубликовать ли технические характеристики шины или сохранить их в тайне.

Если их опубликовать, то другие, так называемые *сторонние,* производители смогут проектировать и продавать платы расширения, совместимые с этой шиной. Доступность дополнительных плат расширения делает компьютер более функциональным, следовательно, на него растет спрос. Рост продаж компьютеров ведет к увеличению рынка для плат расширения. Это побуждает разработчиков большинства небольших компьютерных систем придерживаться принципа *открытой архитектуры*, что позволяет другим производителям создавать периферийные устройства. Со временем шина может превратиться в отраслевой *стандарт*, а стандарты имеют большое значение для индустрии персональных компьютеров.

Самым известным ПК с открытой архитектурой был первый IBM PC, выпущенный осенью 1981 года. Компания IBM опубликовала технический справочник, содержащий полные электрические схемы этого компьютера и всех плат расширения. Этот справочник стал важным инструментом, позволившим многим производителям создать не только свои платы расширения для IBM PC, но и *клоны*, которые были практически идентичны этому компьютеру и использовали то же программное обеспечение.

На долю многочисленных потомков первого компьютера IBM PC в настоящее время приходится 90% рынка*. Несмотря на то что IBM принадлежит лишь небольшая доля этого рынка, она могла бы быть еще меньше, если бы архитектура ее первого компьютера была *закрытой*. Архитектура компьютера Apple Macintosh изначально была закрытой. Несмотря на редкие эксперименты с открытой архитектурой, это принятое в самом начале решение, вероятно, объясняет, почему на долю Macintosh приходится менее 10% рынка настольных ПК. При этом закрытая архитектура компьютерной системы не мешает сторонним компаниям писать для нее *программное обеспечение*. Только

* Ситуация остается таковой и поныне: по данным на январь 2018 года из открытых источников в 2017 году Apple была четвертым по величине производителем компьютеров в мире с долей рынка в районе 7,5–8% (но нужно также учитывать, что «потомка IBM PC» можно не только купить у крупного производителя, но и собрать «на коленке»). *Прим. науч. ред.*

производители некоторых видеоигровых консолей запрещают другим компаниям создавать программы для своих систем.

В первом компьютере IBM PC использовался микропроцессор Intel 8088, позволявший адресовать один мегабайт памяти. Несмотря на то что микропроцессор 8088 — 16-разрядный, обмен данными с памятью он осуществляет фрагментами по восемь бит. Шина, которую компания IBM разработала для своего первого компьютера, теперь носит название ISA (Industry Standard Architecture — архитектура промышленного стандарта). Такая шина предусматривает 62 линии, из которых 20 адресных, восемь используются для ввода и вывода данных, шесть — для запросов на прерывания, три — для запросов на *прямой доступ к памяти* (Direct Memory Access, DMA). Режим DMA позволяет ускорить работу устройств для хранения данных. Обычно чтение и запись данных в память осуществляет микропроцессор. Благодаря режиму DMA другое устройство может перехватить управление шиной и произвести обмен данными непосредственно с памятью, минуя микропроцессор.

В системе S-100 все компоненты размещены на платах расширения. В компьютере IBM PC микропроцессор, некоторые вспомогательные чипы и часть оперативной памяти содержались на плате, которую компания назвала *системной*, хотя эта плата также часто именуется материнской, или главной.

В 1984 году IBM представила персональный компьютер PC/AT, в котором использовался 16-разрядный микропроцессор Intel 80286, позволявший адресовать 16 мегабайт памяти. IBM установила ту же шину, но добавила еще один 36-контактный разъем, который включал семь адресных линий (хотя требовалось всего четыре), восемь линий для ввода и вывода данных, пять линий для запросов на прерывания и четыре линии для запросов на прямой доступ к памяти.

Шины необходимо модернизировать или заменять по мере того, как микропроцессоры перерастают их по разрядности данных (например, при увеличении с 8 до 32 бит), по количеству адресных линий или по быстродействию. Первые шины создавались для микропроцессоров, работавших с тактовой частотой в несколько мегагерц, но точно не в несколько сотен. Когда шина работает на скоростях, для которых не предназначена, она может стать источником высокочастотных помех, вызывающих статический или другой шум в работающих поблизости радиоприемниках и телевизорах.

В 1987 году IBM выпустила шину MCA (Micro Channel Architecture — микроканальная архитектура). Некоторые аспекты этой шины были запатентованы, что позволило компании получать лицензионные платежи. Вероятно, именно по этой причине шина MCA не *стала* отраслевым стандартом. А в 1988 году консорциум из девяти компаний (в который IBM не вошла) изготовил альтернативную 32-разрядную шину EISA (Extended Industry Standard

Architecture — расширенная архитектура промышленного стандарта). В конце 1990-х годов в IBM-совместимых компьютерах широко использовалась разработанная компанией Intel шина PCI (Peripheral Component Interconnect — взаимосвязь периферийных компонентов).

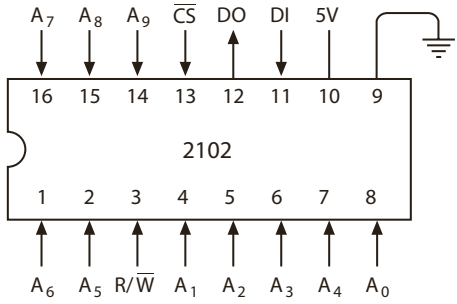
Чтобы понять, как работают различные компоненты компьютера, снова следует вернуться в середину 1970-х, когда все было очень просто. Представим разработку плат для компьютера «Альтаир» или для нашего собственного компьютера на базе процессора 8080 либо 6800. Для него нам, вероятно, потребуется собрать массив RAM, клавиатуру для ввода данных, экран для их вывода, а также некое устройство, позволяющее хранить информацию после выключения. Рассмотрим различные *интерфейсы*, которые можем создать для подключения перечисленных компонентов к нашему компьютеру.

Как вы помните из главы 16, массив RAM предусматривает адресные входы, входы для ввода и вывода данных, а также сигнал, используемый для записи данных в память. От количества адресных входов зависит количество отдельных значений, которые можно сохранить в массиве RAM:

$$\text{Количество значений в массиве RAM} = 2^{\text{количество адресных входов}}.$$

Количество входов для ввода и вывода данных определяет разрядность сохраняемых значений.

В середине 1970-х годов в домашних компьютерах часто применялась микросхема памяти 2102.

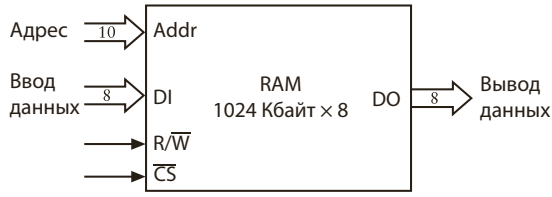


Микросхема 2102 была создана по технологии МОП (металл — оксид — полупроводник), или MOS (metal-oxide-semiconductor), как и сами микропроцессоры 8080 и 6800. Микросхемы МОП можно легко подключить к микросхемам ТТЛ; от последних они отличаются более высокой плотностью транзисторов. К тому же они работают не так быстро.

Если подсчитать количество адресных входов (A_0–A_9) и обратить внимание, что здесь только один вход для ввода (DI) и один вход для вывода (DO) данных, вы поймете, что емкость этой микросхемы ограничена 1024 бит. В зависимости от типа используемой микросхемы 2102 *время доступа*, то есть время, прошедшее между подачей адреса на адресные входы и выводом соответствующих данных на выход DO, составляет от 350 до 1000 наносекунд. При считывании информации из памяти сигнал R/$\overline{W}$ (чтение/запись) обычно равен 1. Когда вам требуется записать данные в микросхему, этот сигнал должен быть равен 0 на протяжении от 170 до 550 наносекунд, опять же в зависимости от типа используемой микросхемы 2102.

Особый интерес представляет сигнал $\overline{CS}$ (Chip Select — выбор микросхемы). Когда этот сигнал равен 1, микросхема *не выбрана*, значит, она не реагирует на сигнал R/$\overline{W}$. Сигнал $\overline{CS}$ оказывает на микросхему и другое важное влияние, о котором расскажу чуть позже.

Разумеется, если вы собираете массив памяти для 8-разрядного микропроцессора, необходимо организовать эту память так, чтобы в ней хранились 8-битные, а не однобитные значения. Для сохранения целых байтов придется объединить по крайней мере восемь таких микросхем, подключив их адресные входы, сигналы R/$\overline{W}$ и $\overline{CS}$ друг к другу. Результат можно изобразить следующим образом.



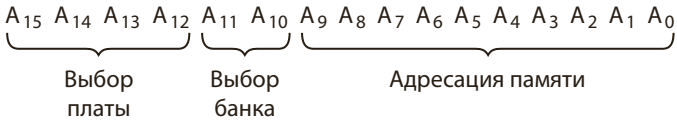
Это массив RAM 1024 × 8 емкостью один килобайт.

С практической точки зрения эти микросхемы памяти нужно монтировать на печатную плату. Сколько их может поместиться на одной плате? Если размещать их максимально близко, то на одну плату S-100 можно установить 64 такие микросхемы, которые обеспечат восемь килобайт памяти.

Попробуем обойтись памятью в четыре килобайта, используя только 32 чипа. Каждый набор микросхем, соединенных между собой для хранения целого байта (как показано выше), называется *банком*. Плата памяти емкостью четыре килобайта содержит четыре банка, каждый из которых состоит из восьми микросхем.

В таких 8-разрядных микропроцессорах, как 8080 и 6800, используются 16-разрядные адреса, с помощью которых можно адресовать 64 килобайта памяти. Когда вы подключаете плату памяти емкостью четыре килобайта,

содержащую четыре банка микросхем, 16 адресных сигналов платы памяти выполняют следующие функции.

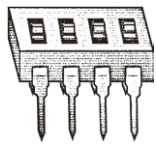


Десять адресных сигналов, с A_0 по A_9, напрямую подключены к микросхемам RAM. Адресные сигналы A_{10} и A_{11} позволяют выбрать, к какому из четырех банков осуществляется обращение. Адресные сигналы с A_{12} по A_{15} определяют, какие адреса относятся к конкретной плате, то есть на какие адреса эта плата реагирует. Наша плата памяти емкостью четыре килобайта может занимать один из шестнадцати 4-килобайтных диапазонов во всем адресном пространстве процессора емкостью 64 килобайта:

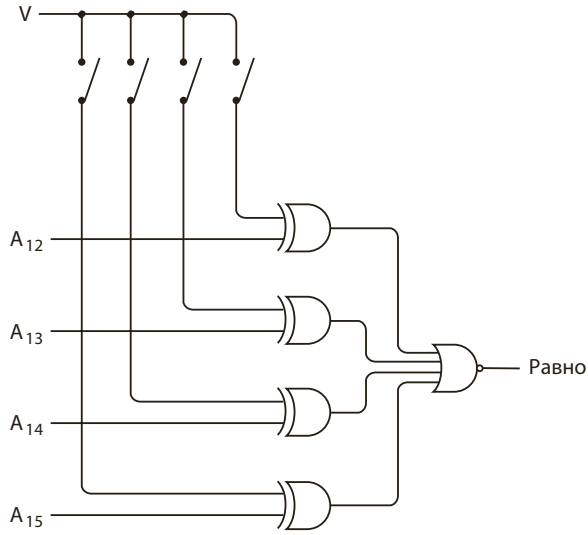
— от 0000h до 0FFFh;
— от 1000h до 1FFFh;
— от 2000h до 2FFFh;
…
— от F000h до FFFFh.

Предположим, мы решили, что к этой плате памяти емкостью четыре килобайта будут относиться адреса в диапазоне от A000h до AFFFh. Значит, адреса с A000h по A3FFh будут заняты первым банком однокилобайтных микросхем, с A400h по A7FFh — вторым, с A800h по ABFFh — третьим, с AC00h по AFFFh — четвертым.

Обычно 4-килобайтная плата памяти предусматривает возможность изменения диапазона адресов, на которые она реагирует. Для этого используется так называемый *DIP-переключатель* (Dual Inline Package), представляющий собой набор крошечных переключателей (от двух до двенадцати) в корпусе с двухрядным расположением выводов, который вставляется в обычное гнездо для интегральной микросхемы.

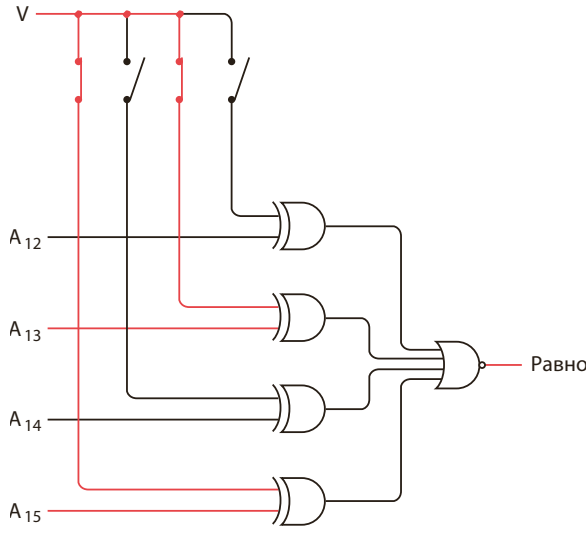


Можно подключить его к четырем старшим адресным разрядам шины, используя схему *компаратор*.

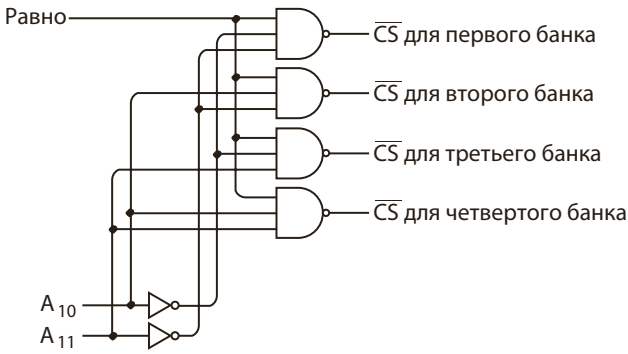


Как вы помните, выход вентиля Искл-ИЛИ равен 1 только тогда, когда на его входы подаются разные значения. Выход вентиля Искл-ИЛИ — 0, если оба входных значения одинаковы.

Например, замыкание переключателей, соответствующих линиям A_{13} и A_{15}, приведет к тому, что плата памяти будет реагировать на адреса с A000h по AFFFh. Когда значения адресных сигналов шины A_{12}, A_{13}, A_{14} и A_{15} равны значениям, установленным с помощью переключателей, выходы всех четырех вентилей Искл-ИЛИ равны 0, значит, выход вентиля ИЛИ-НЕ равен 1.



Затем вы можете объединить этот сигнал «Равно» с дешифратором «2 на 4», чтобы генерировать сигналы $\overline{CS}$ для каждого из четырех банков памяти.



Если сигнал A_{10} равен 0, а A_{11} — 1, значит, выбран третий банк.

Если вы еще помните сложный процесс сборки массивов RAM из главы 16, можете предположить, что нам нужно использовать восемь селекторов «4 на 1» для выбора правильных выходных сигналов от четырех банков памяти. Однако в данном случае они не потребуются, и вот почему.

Как правило, выходные сигналы интегральных схем, совместимых с ТТЛ-чипами, принимают значения либо более 2,2 вольта (логическая единица), либо менее 0,4 вольта (логический ноль). Что произойдет, если вы попытаетесь соединить эти выходные сигналы? Например, к чему приведет соединение выходного сигнала, равного 1, одной схемы и выходного сигнала, равного 0, другой? Определенно ответить на этот вопрос нельзя, поэтому выходы интегральных схем обычно не соединяются друг с другом.

Выходной сигнал микросхемы 2102 известен как сигнал *с тремя состояниями*. Помимо логических 0 и 1, для этого выходного сигнала предусмотрено третье состояние, соответствующее отсутствию какого-либо сигнала, будто этот вывод микросхемы вообще ни к чему не подключен. Выходной сигнал микросхемы 2102 переходит в это третье состояние, когда вход CS равен 1. Это означает, что мы *можем* соединить соответствующие выходные сигналы всех четырех банков и использовать эти восемь комбинированных выходов в качестве восьми линий шины для ввода данных.

Заостряю ваше внимание на выходном сигнале с тремя состояниями, потому что он играет важную роль в работе шины. Практически все платы, подключенные к шине, используют ее линии ввода данных. В любой момент только одна подключенная к шине плата может задействовать эти линии. При этом выходные сигналы остальных плат должны находиться в третьем состоянии.

Микросхема 2102 — это *статическая* память с произвольным доступом, или SRAM (Static Random Access Memory), которая отличается от *динамической* памяти с произвольным доступом, или DRAM (Dynamic Random Access Memory). Памяти SRAM обычно требуется четыре транзистора для хранения одного бита (это не так много, как в триггерах из главы 16). Памяти DRAM для этого нужен только один транзистор. Однако недостаток памяти DRAM — необходимость использования более сложных вспомогательных схем.

Содержимое памяти SRAM, например микросхемы 2102, сохраняется только при наличии питания. Если питание отключается, содержимое исчезает. Это касается и памяти DRAM, однако микросхема DRAM также требует периодического считывания данных, даже если в них нет необходимости. Такой цикл *обновления* должен повторяться несколько сотен раз в секунду. Это все равно что периодически тормошить человека, чтобы он не заснул.

Несмотря на сложности, связанные с использованием памяти DRAM, постоянно увеличивающаяся емкость этих микросхем сделала их стандартом. В 1975 году компания Intel представила микросхему DRAM емкостью 16 384 бит. В соответствии с законом Мура емкость микросхем DRAM увеличивается в четыре раза каждые три года. Современные компьютеры обычно предусматривают гнезда для памяти прямо на системной плате, куда вставляются небольшие платы, называемые *модулями памяти* SIMM (Single Inline Memory Module) или DIMM (Dual Inline Memory Module), с несколькими микросхемами DRAM.

Теперь вы знаете, как создавать платы памяти, однако не стоит заполнять памятью все адресное пространство микропроцессора. Нужно выделить некоторую его часть для устройства вывода.

Электронно-лучевая трубка (ЭЛТ) была наиболее распространенным устройством вывода для компьютеров. ЭЛТ, подключенная к компьютеру, обычно называется *дисплеем,* или *монитором*; электронный компонент, подающий дисплею сигнал, именуется *видеоадаптером*. Часто видеоадаптер занимает в компьютере отдельную *видеокарту*.

Несмотря на то что двумерное изображение дисплея или телевизора может показаться сложным, на самом деле оно создается одним непрерывным лучом света, который быстро пробегает по экрану. Луч начинает свое движение в верхнем левом углу и перемещается по экрану вправо, после чего возвращается к левому краю, чтобы начать следующую горизонтальную линию, известную как *строка развертки*. Перемещение луча справа налево к началу очередной строки — *обратный ход по горизонтали*. Когда луч заканчивает последнюю строку, он возвращается из нижнего правого в верхний левый угол экрана (*обратный ход по вертикали*), и весь процесс начинается снова.

По американским стандартам телевизионного вещания это должно происходить 60 раз в секунду. Такая *частота развертки* достаточно высока, чтобы изображение не мерцало.

Телевизор устроен сложнее из-за использования *чересстрочной развертки*. Один *кадр* — отдельное неподвижное изображение — разбивается на два поля из четных строк (первое) и нечетных (второе). *Частота строчной развертки*, или частота следования строк, составляет 15 750 герц. Если разделить это число на 60 герц, получим 262,5 строки в одном поле кадра. Целый кадр содержит в два раза больше, то есть 525 строк развертки*.

Вне зависимости от вида развертки дисплея непрерывный луч света, порождающий видеоизображение, управляется одним непрерывным сигналом. Хотя аудио- и видеосигналы телевизионной программы объединяются при трансляции или передаче через систему кабельного телевидения, в итоге они разделяются. *Видеосигнал*, который я опишу, идентичен сигналу, подающемуся на соответствующие входы и выходы видеомагнитофонов, камер и некоторых телевизоров.

В случае черно-белого телевидения этот видеосигнал достаточно прост и понятен. (Цвет все усложняет.) Шестьдесят раз в секунду в этот сигнал записывается *импульс вертикальной синхронизации*, который указывает на начало поля. Этот импульс — подача нулевого напряжения (земля) на протяжении 400 микросекунд. *Импульс горизонтальной синхронизации* указывает на начало каждой строки развертки: 15 750 раз в секунду в течение пяти микросекунд подается нулевое напряжение. Между импульсами горизонтальной синхронизации значение сигнала варьируется от 0,5 вольта (черный) до двух вольт (белый), оттенки серого соответствуют значениям от 0,5 до 2 вольт**.

Таким образом, изображение телевизора является отчасти цифровым, отчасти аналоговым. Это изображение разделено по вертикали на 525 строк, однако внутри каждой из строк развертки напряжение непрерывно меняется, что обусловливает вариации яркости. Однако напряжение не может изменяться произвольно. Существует предельная скорость, с которой телеэкран может реагировать на изменение сигнала. Этот показатель называется «пропускная способность».

---

* Для американского стандарта NTSC. В стандарте SECAM, применяемом в России, используются 625 строк, в стандарте PAL — 405. К слову, американский стандарт NTSC оказался весьма чувствительным к амплитудно-частотным искажениям сигнала, приводящим к искажениям цвета, за что получил шуточную расшифровку Never The Same Color («каждый раз другого цвета»). *Прим. науч. ред.*

** Кроме того, существует уровень «чернее черного» (ноль вольт), на котором передаются импульсы синхронизации. *Прим. науч. ред.*

Пропускная способность — чрезвычайно важная концепция в сфере связи, поскольку определяет, какой объем информации можно передать по конкретному каналу. В случае телевизора пропускная способность — предельная скорость, с которой видеосигнал может измениться от уровня черного до уровня белого, а затем снова до уровня черного. По американским стандартам телевизионного вещания полоса пропускания примерно равна 4,2 мегагерца*.

Когда дисплей необходимо подключить к компьютеру, его неудобно представлять в виде гибрида аналогового и цифрового устройства. Его легче рассматривать как полностью цифровое устройство. С точки зрения компьютера видеоизображение удобнее отобразить как прямоугольную сетку с дискретными точками, называемыми *пикселами* (pixel от picture element — «элемент изображений»).

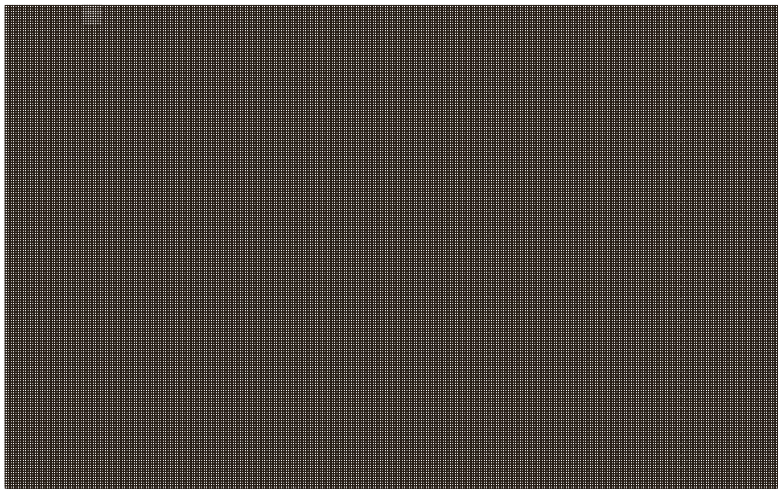
Полоса пропускания видеодисплея ограничивает количество пикселов в горизонтальной строке развертки. Я определил полосу пропускания как скорость, с которой видеосигнал может измениться от уровня черного до уровня белого, а затем снова до уровня черного. Полоса пропускания телевизоров, равная 4,2 мегагерца, допускает создание двух пикселов 4,2 миллиона раз в секунду. Если разделить произведение 2 × 4 200 000 на 15 750 (частота строчной развертки), получим 533 пиксела в каждой строке. Однако примерно треть этих пикселов оказывается невидимой из-за их нахождения на дальних концах изображения или из-за обратного хода луча по горизонтали. Таким образом, каждая строка — это примерно 320 пикселов.

Точно так же по вертикали не насчитывается 525 пикселов. Некоторые из них теряются в верхней и нижней части экрана и во время обратного хода луча по вертикали. Кроме того, *не следует* полагаться на чересстрочную развертку при использовании телевизора в качестве дисплея. Таким образом, реальное количество пикселов по вертикали равно приблизительно двумстам.

Итак, можно сказать, что *разрешение* примитивного видеоадаптера, подключенного к обычному телевизору, составляет 320 пикселов по горизонтали на 200 пикселов по вертикали, или 320 × 200.

---

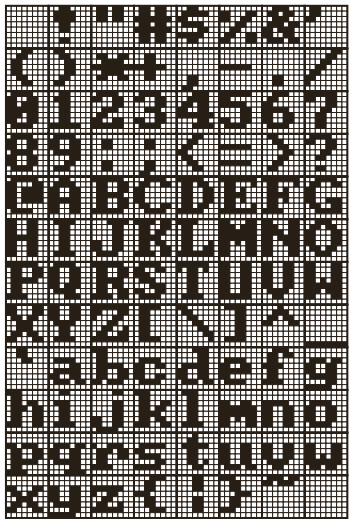
* Полоса пропускания и пропускная способность — это разные вещи. Первое — это полоса частот (ширина частотного спектра), в которой осуществляется передача сигнала, второе — это максимальная скорость передачи сигнала в битах в секунду. В России полоса пропускания аналогового телевидения составляет порядка шести мегагерц. *Прим. науч. ред.*



Чтобы определить общее количество пикселов в этой сетке, можете подсчитать их или просто перемножить числа 320 и 200, получив 64 тысячи пикселов. В зависимости от того, как вы сконфигурировали видеоадаптер (о чем расскажу чуть позже), каждый пиксел может быть либо черным, либо белым (черно-белое изображение), либо обладать определенным цветом (цветное изображение).

Предположим, нам нужно отобразить на этом дисплее некоторый текст. Сколько текста может на нем уместиться?

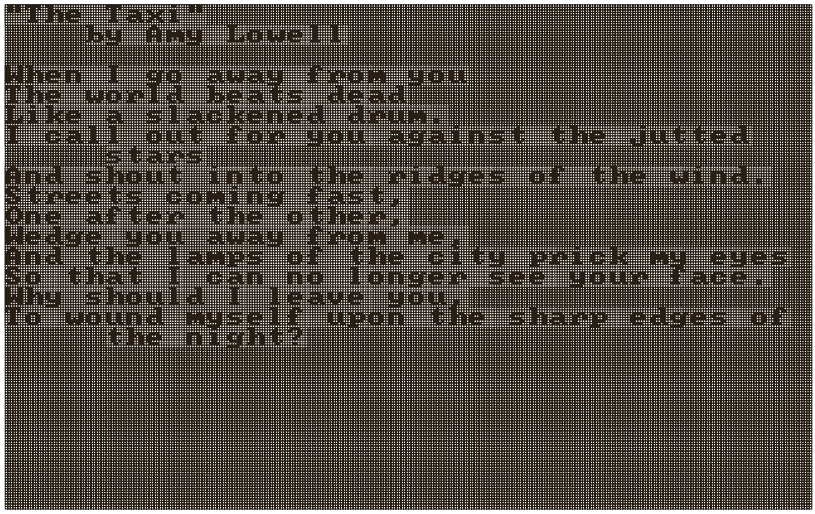
Очевидно, все зависит от того, сколько пикселов используется для отображения каждого текстового символа. Далее представлен один из возможных подходов, при котором для каждого символа используется сетка 8 × 8 (64 пиксела).



Эти символы соответствуют кодам ASCII с 20h по 7Fh. Коды ASCII с 00h до 1Fh отведены под неотображаемые символы.

В приведенном примере каждый символ соответствует не только 7-битному ASCII-коду, но и 64 битам на экране, определяющим его внешний вид. Эти 64 бита тоже можно рассматривать в качестве своеобразного кода.

При таком способе представления символов вы можете уместить на видеодисплее с разрешением 320 × 200 пикселов 25 строк по 40 символов. Этого достаточно, например, для короткого стихотворения Эми Лоуэлл.



Видеоадаптер потребляет некоторое количество оперативной памяти для хранения содержимого дисплея, а микропроцессор должен иметь возможность записывать данные в эту память для изменения изображения на этом дисплее. Удобнее всего, когда память RAM является частью общего адресного пространства процессора. Сколько же оперативной памяти потребуется видеоадаптеру, который я описываю?

Это непростой вопрос! Значение может варьироваться от 1 до 192 килобайт!

Сначала оценим нижний предел. Один из способов уменьшения требований к памяти заключается в том, чтобы ограничить возможности адаптера только отображением текста. Мы уже выяснили, что можем отобразить 25 строк по 40 символов, или 1000 символов. В памяти RAM на видеоплате должны храниться только 7-битные ASCII-коды соответствующих символов. Тысяча 7-битных значений — приблизительно 1024 байт, или один килобайт.

Такая плата видеоадаптера также должна оснащаться *генератором символов*, содержащим точечные шаблоны всех символов ASCII — вроде тех, которые были показаны на одном из предыдущих изображений. Как правило,

генератор символов — это *постоянное запоминающее устройство,* или ПЗУ (Read-Only Memory, ROM), интегральная схема, изготовленная таким образом, что в ответ на обращение к конкретному адресу всегда выдаются одни и те же данные. В отличие от памяти RAM, ПЗУ не предусматривает никаких сигналов для ввода данных.

Память ПЗУ можно считать схемой, преобразующей один код в другой. ПЗУ, в котором хранятся точечные шаблоны (8 × 8 пикселов) для 128 символов ASCII, может предусматривать семь адресных входов (для ASCII-кодов) и 64 выхода для данных. Таким образом, ПЗУ преобразует 7-битный ASCII-код в 64-битный, определяющий внешний вид символа. Однако наличие 64 выходов сделали бы чип слишком громоздким! Гораздо удобнее использовать десять адресных входов и восемь выходов. Семь адресных сигналов указывают на конкретный ASCII-символ. (Эти семь битов адреса подаются с выходов RAM на видеоплате.) Другие три адресных сигнала определяют строку. Например, биты адреса 000 соответствуют верхней строке точечного шаблона, а биты 111 — нижней строке, восемь выходных битов — восьми пикселам каждой строки.

Предположим, что ASCII-код равен 41h. Этот код сопоставим с заглавной буквой *A*. Ее точечный шаблон состоит из восьми строк по восемь бит. В следующей таблице приведены 10-битные адреса (пробел отделяет ASCII-код от кода строки) и сигналы на выходах для данных, соответствующие заглавной букве *A*.

Адрес	Сигнал на выходе
1000001 000	00110000
1000001 001	01111000
1000001 010	11001100
1000001 011	11001100
1000001 100	11111100
1000001 101	11001100
1000001 110	11001100
1000001 111	00000000

Вы видите букву *A*, нарисованную единицами на фоне нулей?

Видеоадаптер, воссоздающий только текст, также должен предусматривать возможность отображения *курсора* — черточки в том месте экрана, где появится следующий введенный с клавиатуры символ. Номера строки и столбца, совпадающие с позицией курсора, обычно хранятся в двух 8-битных регистрах на видеоплате, в которые микропроцессор может записывать значения.

Если плата видеоадаптера не ограничивается воспроизведением текста, она называется *графической*. Записывая данные в оперативную память графической

платы, микропроцессор может выводить на экран изображения, а также отображать текст разных размеров и стилей. Графическим платам требуется больше памяти, чем текстовым. При разрешении 320 × 200 изображение состоит из 64 тысяч пикселов. Если каждый пиксел соответствует одному биту памяти, такой плате требуется 64 тысячи бит, или 8000 байт, оперативной памяти. Это, конечно, абсолютный минимум. То, что один бит памяти равен одному пикселу, позволяет использовать только два цвета, например черный и белый. Значение ноль бит может соотноситься с черным пикселом, а один бит — с белым.

Разумеется, черно-белые телевизоры отображают не только черные и белые пикселы. Они также способны воспроизводить множество оттенков серого. Чтобы графическая плата могла передать оттенки серого, для каждого пиксела обычно отводится целый *байт* памяти RAM, причем значение 00h соответствует черному цвету, FFh — белому, а все промежуточные значения — разнообразным оттенкам серого. Видеоплате, отображающей 256 оттенков серого на дисплее с разрешением 320 × 200, требуется 64 тысячи *байт* памяти. Это почти все адресное пространство одного из 8-битных микропроцессоров, о которых я говорил ранее!

Использование полноцветного режима предполагает выделение трех байтов на каждый пиксел. Если вы рассмотрите экран телевизора или компьютерного дисплея через увеличительное стекло, то обнаружите, что каждый цвет создается различными комбинациями основных цветов: красного, зеленого и синего. Чтобы отобразить весь диапазон, требуется по одному байту для указания интенсивности каждого из трех основных цветов, то есть 192 тысячи байт памяти RAM. (В последней главе я подробно остановлюсь на графике.)

Количество различных цветов, которые способен отобразить видеоадаптер, определяется количеством битов, выделенных для каждого пиксела. Это соотношение может показаться знакомым, поскольку в нем используется степень двойки:

$$\text{Количество цветов} = 2^{\text{количество битов на пиксел}}.$$

Разрешение 320 × 200 пикселов максимально для типичного телевизора. Именно поэтому мониторы, созданные специально для компьютеров, имеют гораздо более широкую полосу пропускания, чем телевизионные экраны. Первые мониторы, которые продавались с компьютером IBM PC в 1981 году, могли отображать 25 строк по 80 символов. Именно такое количество символов воспроизводилось дисплеями CRT, которые использовались с большими и дорогими мейнфреймами IBM. Для компании IBM число 80 имеет особое значение. Почему? *Именно столько символов умещалось на перфокарте IBM!* Действительно, поначалу дисплеи CRT, подключенные к мейнфреймам, часто

использовались для просмотра содержимого перфокарт. Даже сегодня кое-кто продолжает называть строки дисплея, отображающего только текст, *картами*.

На протяжении многих лет видеоадаптеры совершенствовались в плане разрешения и цветопередачи. Важная веха была достигнута в 1987 году, когда в персональных компьютерах IBM серии Personal System/2 и Apple Macintosh II начали применяться видеоадаптеры, способные отображать 640 пикселов по горизонтали и 480 пикселов по вертикали. С тех пор этот показатель — минимальное стандартное разрешение для видео.

Это может показаться невероятным, но причина важности разрешения 640 × 480 связана с работой Томаса Эдисона! Примерно в 1889 году, когда Эдисон и его инженер Уильям Диксон работали над кинокамерой «Кинетограф» и проектором «Кинетоскоп», они решили сделать так, чтобы ширина движущегося изображения на треть превосходила его высоту. Соотношение ширины и высоты изображения называется *характеристическим отношением*. Соотношение, которое выбрали Эдисон и Диксон, обычно выражается как 1,33 : 1, или 4 : 3. На протяжении более 60 лет это соотношение использовалось при производстве кинофильмов и конструировании телевизоров. Только в начале 1950-х голливудские студии начали снимать фильмы в широкоэкранном формате, что и составило конкуренцию телевидению благодаря выходу за рамки соотношения 4 : 3.

Большинство компьютерных мониторов (и телевизоров) имеет характеристическое отношение 4 : 3*, в чем вы можете убедиться с помощью линейки. Разрешение 640 × 480 также соответствует отношению 4 : 3. Это значит, что горизонтальная линия, состоящая, например, из 100 пикселов, имеет ту же физическую длину, что и вертикальная линия из 100 пикселов. Таким образом, пикселы являются *квадратными*, что считается предпочтительным для компьютерной графики.

Видеоадаптеры и мониторы практически всегда имеют разрешение 640 × 480, однако они также способны работать в видеорежимах с разрешением 800 × 600, 1024 × 768, 1280 × 960 и 1600 × 1200.

Обычно компьютерный дисплей и клавиатура кажутся нам связанными, поскольку символы, введенные с клавиатуры, отображаются на экране. Однако на самом деле они, как правило, не зависят друг от друга.

Каждая клавиша на клавиатуре, по сути, простой переключатель. При нажатии клавиши переключатель замыкается. Первые клавиатуры напоминали пишущую машинку и состояли всего из 48 клавиш; клавиатура современных персональных компьютеров часто насчитывает более 100 клавиш.

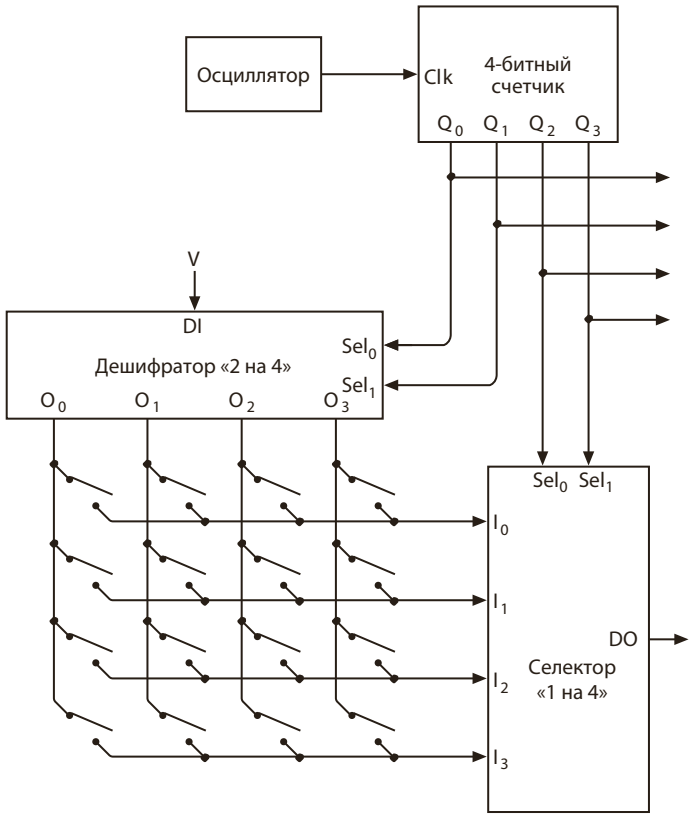
---

* Сегодня соотношение 16 : 9 (1920 × 1080) распространено шире. *Прим. науч. ред.*

Клавиатура, подключенная к компьютеру, должна предусматривать некоторое оборудование, предоставляющее уникальный код для каждой нажатой клавиши. Вы можете предположить, что этот код является ASCII-кодом, соответствующим символу на клавише. Однако разрабатывать аппаратные средства, определяющие ASCII-код, непрактично и нежелательно. Например, клавиша A может соответствовать ASCII-коду 41h или 61h в зависимости от того, нажимается ли вместе с ней клавиша Shift. Кроме того, на современных компьютерных клавиатурах есть множество клавиш, которые не соответствуют символам ASCII. Код, предоставляемый аппаратным обеспечением клавиатуры, называется *скан-кодом*. Для определения соответствия нажатой клавиши какому-либо ASCII-коду используется небольшая компьютерная программа.

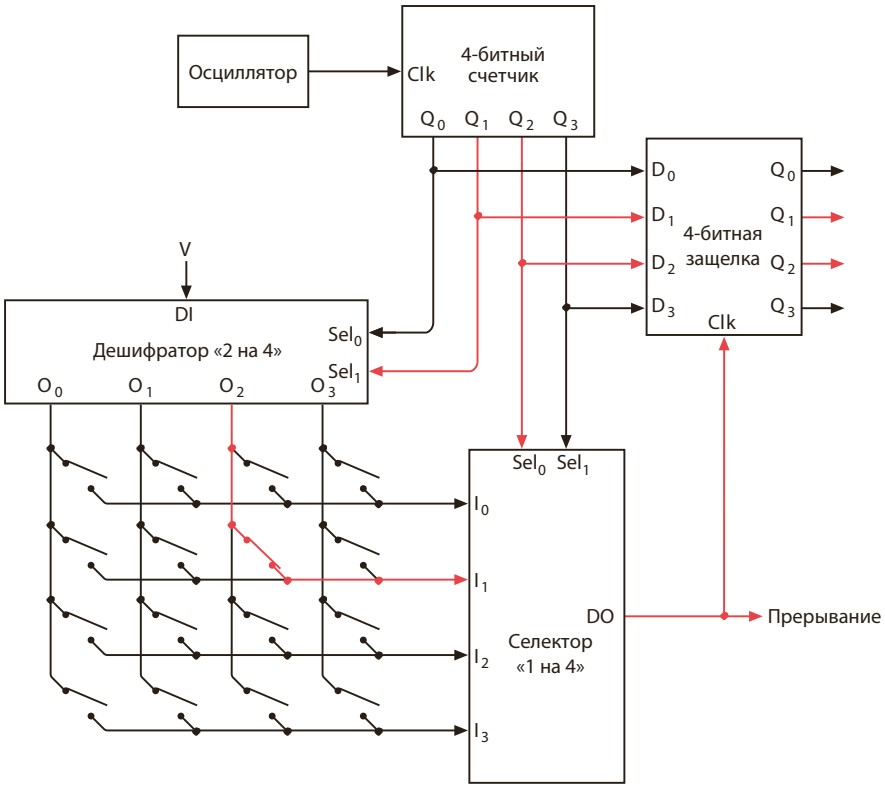
Чтобы схема аппаратного обеспечения клавиатуры не стала чересчур запутанной, будем считать, что она состоит всего из 16 клавиш. При нажатии клавиши аппаратное обеспечение должно сгенерировать 4-битный код, принимающий двоичные значения в диапазоне от 0000 до 1111.

Аппаратное обеспечение клавиатуры содержит уже знакомые нам компоненты.



Шестнадцать клавиш представлены в виде простых переключателей в нижней левой части схемы. Четырехбитный счетчик многократно и быстро перебирает 16 кодов, соответствующих клавишам. Он должен делать это быстрее, чем человек нажимает и отпускает клавишу.

Выходные сигналы 4-битного счетчика подаются на входы Sel дешифратора «2 на 4» и селектора «4 на 1». Если не была нажата ни одна клавиша, ни один из входов селектора не будет равен 1. Следовательно, и выход не будет равен 1. Однако если нажата конкретная клавиша, то при соответствующем ей значении выходного сигнала 4-битного счетчика выход селектора будет равен 1. Например, при нажатии второй сверху и справа клавиши и выходном сигнале счетчика 0110 выход селектора будет равен 1.



Это код, соответствующий данной клавише. Когда клавиша нажата, ни одно другое значение выходного сигнала счетчика не приведет к тому, что выход селектора станет равным 1. Для каждой клавиши предусмотрен собственный код.

Если клавиатура состоит из 64 клавиш, вам понадобится 6-битный сканкод и 6-битный счетчик. Вы можете организовать клавиши в массив 8 × 8, используя дешифратор «3 на 8» и селектор «1 на 8». Если клавиатура имеет от 65

до 128 клавиш, потребуется 7-битный код. Клавиши можно организовать в массив 8 × 16 и использовать дешифратор «4 на 16» и селектор «8 на 1» (или дешифратор «3 на 8» и селектор «16 на 1»).

То, что происходит дальше в этой схеме, зависит от сложности интерфейса клавиатуры. Аппаратное обеспечение может предусматривать для каждой клавиши один бит оперативной памяти. Память RAM будет адресоваться счетчиком, а содержимым этой памяти может стать 0, если клавиша не нажата, и 1 — если нажата. Эту память RAM также может считывать микропроцессор для определения состояния каждой из клавиш.

Одна из полезных функций интерфейса клавиатуры — сигнал прерывания. Как вы помните, микропроцессор 8080 предусматривает входной сигнал, который позволяет внешнему устройству прерывать работу микропроцессора. В ответ на этот сигнал процессор считывает команду из памяти. Обычно это команда RST, заставляющая процессор перейти к определенной ячейке, где хранится программа для обработки прерывания.

Последнее периферийное устройство, которое опишу в этой главе, — устройство для долговременного хранения данных. Как вы помните, оперативная память вне зависимости от того, из чего она собрана (реле, вакуумных ламп или транзисторов), теряет свое содержимое при отключении питания. По этой причине компьютеру требуется устройство для долговременного хранения данных. Один проверенный временем способ — пробивание отверстий в бумажных или картонных картах, вроде перфокарт IBM. На заре эры небольших компьютеров для сохранения и последующей загрузки программ и данных отверстия пробивались в рулонах бумажной ленты.

Недостатки перфокарт и бумажной ленты в том, что их нельзя использовать повторно. Пробитое отверстие непросто заклеить. Еще один дефект — невысокая эффективность. Сейчас, если вы можете *увидеть* бит невооруженным глазом, можете с уверенностью сказать: «Он занимает слишком много места!»

По этим причинам гораздо чаще встречаются *магнитные накопители*. Принцип их работы был описан еще в 1878 году американским инженером Оберлином Смитом (1840–1926). Однако первое работающее устройство было создано только 20 лет спустя, в 1898 году, датским изобретателем Вальдемаром Поульсеном (1869–1942). Поначалу «телеграфон» Поульсена предназначался для записи телефонных сообщений, если никто не мог взять трубку. Для записи звука на стальной проволоке применялся электромагнит — вездесущее устройство, с которым мы познакомились, когда рассматривали телеграф. Электромагнит намагничивал проволоку в соответствии с изменениями формы звуковой волны, а для воспроизведения звука проволока с той же скоростью протягивалась вдоль обмоток электромагнита, индуцируя в них ток. Электромагнит,

созданный для хранения и считывания информации, называется *головкой* вне зависимости от типа магнитного накопителя.

В 1928 году австрийский изобретатель Фриц Пфлеумер (1881–1945) запатентовал магнитное записывающее устройство, в котором использовалась бумажная лента с металлическим напылением, сделанная по технологии, разработанной для металлизированных полосок на сигаретах. Вскоре бумагу заменила более прочная ацетилцеллюлозная основа, благодаря чему родился один из самых надежных и хорошо известных носителей информации. Магнитная лента, теперь упакованная в пластиковые кассеты, нашла применение в записи и воспроизведении музыки и видео.

Первая коммерческая система для записи цифровых компьютерных данных на магнитную ленту была представлена компанией Remington Rand в 1950 году. В то время на катушке ленты шириной в полдюйма (1,27 сантиметра) могло поместиться несколько мегабайт. На заре эры домашних компьютеров люди превращали обычные кассетные магнитофоны в устройства для записи. С помощью небольших программ содержимое блока памяти записывалось на ленту, а позднее считывалось с нее. Первые компьютеры IBM PC предусматривали разъем для кассетного накопителя. Магнитная лента употреблялась для долгосрочного архивирования данных. Тем не менее этот носитель не идеален из-за невозможности быстрого перехода к нужному месту на ленте. Обычно для этого требуется перемотать ее вперед или назад, а это занимает время.

Носителем, обеспечивающим более быстрый доступ к данным, является диск. Сам диск вращается вокруг своей оси, пока над ним перемещается штанга с одной или несколькими головками, благодаря чему доступ к любой области диска осуществляется очень быстро.

Магнитные диски фактически использовались для звукозаписи еще до магнитной ленты. Однако первый диск для хранения компьютерных данных был изобретен в компании IBM в 1956 году. Дисковая система памяти RAMAC (Random Access Method for Accounting and Control) содержала 50 металлических дисков диаметром 60 сантиметров и могла хранить пять мегабайт.

С тех пор размер дисков значительно уменьшился, а емкость увеличилась. Диски обычно делятся на *гибкие* (floppy, *дискеты*) и *жесткие* (hard, *несъемные диски*). Дискета — это пластиковый диск, заключенный в корпус (корпус поначалу делали картонным, затем — пластиковым). Пластиковый корпус не дает дискете гнуться, поэтому она уже не такая *гибкая*, как раньше, хотя и продолжает называться гибким диском. Для записи и чтения данных с дискеты ее необходимо поместить в специальное устройство под названием *флоппи-дисковод*. Диаметр первых гибких дисков составлял около 20 сантиметров. В первых компьютерах IBM PC устанавливались гибкие диски диаметром около

13 сантиметров; затем дискеты диаметром около девяти сантиметров. Возможность извлекать эти гибкие диски из дисковода позволяет с их помощью переносить данные с одного компьютера на другой. Кроме того, на дискетах распространялось коммерческое программное обеспечение.

Жесткий диск обычно состоит из нескольких металлических дисков, встроенных в дисковод. Как правило, жесткие диски работают быстрее и вмещают больше данных, чем дискеты, однако их невозможно извлечь.

Поверхность диска разделена на концентрические кольца, называемые *дорожками*. Каждая дорожка разделена на *сектора*, которые хранят определенное количество данных, обычно 512 байт. Флоппи-дисковод первого компьютера IBM PC использовал лишь одну сторону 13-сантиметровой дискеты и разделял ее на 40 дорожек по восемь секторов, каждый из которых хранил 512 байт. Таким образом, на каждой дискете находились 163 840 байт, или 160 килобайт. Дискеты 3,5 дюйма, использовавшиеся в PC-совместимых компьютерах, имели две стороны по 80 дорожек и по 18 секторов на дорожку. Каждый сектор такой дискеты хранил 512 байт, что обеспечивало общую емкость в 1 474 560 байт, или 1440 килобайт.

Емкость первого жесткого диска, представленного в 1983 году IBM в компьютере PC/XT, составляла десять мегабайт. В 1999 году менее чем за 400 долларов можно было приобрести жесткий диск емкостью 20 гигабайт (20 *миллиардов* байт).

Как правило, дискета или жесткий диск предусматривает собственный электронный интерфейс, однако для обмена данными с микропроцессором требуется еще один. Наиболее популярные стандарты интерфейсов для жестких дисков — SCSI (Small Computer System Interface), ESDI (Enhanced Small Device Interface) и IDE (Integrated Device Electronics)*. Все эти интерфейсы используют прямой доступ к памяти (DMA) для того, чтобы перехватить управление шиной и осуществлять обмен данными непосредственно между оперативной памятью и диском, минуя микропроцессор. При этом обмен информацией происходит фрагментами, соответствующими размеру дискового сектора, который обычно равен 512 байт.

Многие начинающие пользователи домашних компьютеров, наслушавшись разговоров о мегабайтах и гигабайтах, начинают путать полупроводниковую оперативную память с жестким диском. В последние годы появилось правило, позволяющее избежать путаницы в терминологии. Согласно ему слово «память» следует использовать только для обозначения полупроводниковой оперативной памяти, а термины «накопитель» и «запоминающее устройство» — для

* За прошедшие с момента написания книги годы главенствующее место занимали IDE, EIDE, а теперь SATA. *Прим. науч. ред.*

обозначения всего остального, то есть дискет, жестких дисков и магнитной ленты. В этой книге я старался придерживаться этого принципа.

Наиболее очевидное различие между памятью и накопителем в том, что память является энергозависимой: она теряет свое содержимое при отключении питания. Накопитель не зависит от питания. Данные сохраняются на дискете или жестком диске до тех пор, пока пользователь не сотрет их или не перезапишет. Тем не менее существует еще одно значительное различие, которое можно заметить, только поняв принцип работы микропроцессора. При подаче адресного сигнала микропроцессор всегда обращается к памяти, а не к запоминающему устройству.

Перемещение данных из накопителя в память для их последующего использования микропроцессором требует дополнительных действий. Для этого микропроцессор должен выполнить небольшую программу, которая осуществляет обращение к диску.

Чтобы понять разницу между памятью и накопителем, можно использовать следующую аналогию: память похожа на рабочий стол. Вы можете работать со всем, что находится на столе. Накопитель подобен шкафу с папками. Если нужна какая-то из папок, вы должны встать, подойти к шкафу, достать нужную и положить ее на стол. Когда на вашем столе оказывается слишком много папок, нужно убрать некоторые из них обратно в шкаф.

Данные на диске хранятся в виде так называемых *файлов*. За сохранение и извлечение файлов отвечает чрезвычайно важная программа — *операционная система*.

Глава 22
Операционная система

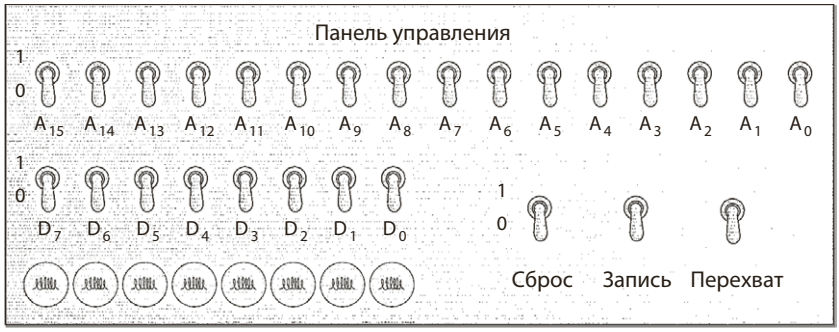
Наконец мы собрали нечто напоминающее полноценный компьютер, по крайней мере мысленно. Он оснащен микропроцессором, оперативной памятью, клавиатурой, монитором и жестким диском. Все оборудование на месте, и мы с нетерпением смотрим на кнопку включения, которая его оживит. Вероятно, этот проект напомнил, как Виктор Франкенштейн собирал своего монстра, как папа Карло строгал Буратино.

И все же нам чего-то не хватает. И отнюдь не молнии, не заклинаний. Включите этот новый компьютер и скажите, что вы видите.

По мере разогрева электронно-лучевой трубки на экране отобразится массив идеально оформленных, но совершенно случайных символов ASCII. Это мы и ожидали. Полупроводниковая память теряет свое содержимое при отключении питания, а при следующем включении приходит в случайное и непредсказуемое состояние. Вся созданная нами для микропроцессора память RAM — случайный набор байтов. Микропроцессор воспринимает этот набор как машинный код и начинает с ним работать, что не приведет к *нежелательным* последствиям, в частности компьютер не взорвется, — однако не будет и пользы.

Нам не хватает программного обеспечения. При первом включении или перезагрузке микропроцессор обращается к машинному коду, начиная с ячейки памяти с определенным адресом. В случае Intel 8080 это 0000h. При включении правильно спроектированного компьютера адрес памяти должен содержать машинную инструкцию (вероятно, первую из многих).

Как эта инструкция туда попадает? Процесс записи программного обеспечения в новый компьютер является, вероятно, одним из наиболее запутанных этапов. Один из способов решения этой задачи предполагает использование пульта управления, аналогичного описанному в главе 16, где он применялся для записи байтов в оперативную память и их последующего считывания.



В отличие от описанного ранее пульта управления, этот имеет переключатель «Сброс», который подключен к одноименному входу микропроцессора. Пока этот переключатель замкнут, микропроцессор ничего не делает. После размыкания переключателя микропроцессор обращается к машинному коду.

Чтобы использовать этот пульт управления, замкните переключатель «Сброс» для остановки микропроцессора, а переключатель «Перехват» — для перехвата управления адресными линиями и линиями данных шины. Теперь вы можете использовать переключатели с A_0 по A_{15} для указания 16-разрядного адреса памяти. Лампочки с D_0 по D_7 показывают 8-битное содержимое этого адреса. Чтобы записать сюда новое значение, введите нужный адрес с помощью переключателей с D_0 по D_7, а затем последовательно замкните и разомкните переключатель «Запись». Закончив ввод новых данных в память, разомкните переключатели «Перехват» и «Сброс», и микропроцессор начнет выполнять программу.

Таким образом следует вводить первые машинные инструкции в только что собранный компьютер. Разумеется, это потребует больших усилий. Безусловно, время от времени вы будете ошибаться. Пальцы покроются мозолями, мозги закипят, но это издержки профессии.

Когда на дисплее увидите результаты работы своих программ, все усилия будут вознаграждены. Текстовый дисплей, который мы собрали в предыдущей главе, имеет видеопамять емкостью один килобайт для хранения текстов в кодировке ASCII объемом 25 строк по 40 символов каждая. Программа записывает данные в эту память так же, как в любую другую.

Однако вывести результат программы на дисплей не так просто, как может показаться. Например, если ваша программа совершает некоторые вычисления, в результате которых получается 4Bh, вы не можете просто записать это значение в видеопамять. В этом случае на экране отобразилась бы буква *K*, поскольку именно ей соответствует код ASCII 4Bh. Вместо этого вам нужно записать в видеопамять *два* символа ASCII: 34h — код ASCII для цифры 4

и 42h — код ASCII для буквы *B*. Каждая тетрада 8-битного результата — шестнадцатеричное значение, которое должно отображаться на экране.

Вероятно, придется написать небольшую подпрограмму для выполнения этого преобразования. Приведем одну из таких подпрограмм на языке ассемблера для микропроцессоров 8080, которая преобразует записанную в аккумуляторе тетраду (предполагается, что это значение находится в диапазоне от 00h до 0Fh включительно) в соответствующий ей код ASCII.

```
NibbleToAscii: CPI A,0Ah; Проверяем, буква или цифра
               JC Number
               ADD A,37h; Цифры A-F преобразуем в 41h-46h
               RET
Number:        ADD A,30h; Цифры 0-9 преобразуем в 30h-39h
               RET
```

Следующая подпрограмма дважды вызывает подпрограмму NibbleToAscii для преобразования байта, записанного в аккумуляторе A, в две цифры ASCII и их записи в регистры B и C.

```
ByteToAscii:   PUSH PSW  ; Сохраняем аккумулятор
               RRC              ; Четырежды сдвигаем содержимое A,
               RRC              ; чтобы добраться до старшей
               RRC              ; тетрады
               RRC
               CALL NibbleToAscii; Преобразуем в ASCII код
               MOV B, A    ; Перемещаем результат в регистр B
               POP PSW     ; Возвращаем содержимое аккумулятора
               AND A,0Fh; Получаем младшую тетраду
               CALL NibbleToAscii; Преобразуем в ASCII код
               MOV A, C    ; Перемещаем результат в регистр C
               RET
```

Эти подпрограммы теперь позволяют отобразить на дисплее байт в шестнадцатеричном формате. Если захотите преобразовать значение в десятичный формат, потребуется еще немного поработать. На самом деле этот процесс похож на то, как люди переводят значение из шестнадцатеричной системы счисления в десятичную путем многократного деления на 10.

Помните: фактически вы не вводите в память эти программы на ассемблере. Вместо этого вы пишете их на бумаге, а затем преобразуете в машинный

код, который вводите в память. Мы продолжим заниматься такой «ручной сборкой» вплоть до главы 24.

Несмотря на то что с аппаратной точки зрения пульт управления прост, им неудобно пользоваться. Это наихудший из когда-либо разработанных способов ввода и вывода данных. Досадно, что нам хватило ума собрать с нуля собственный компьютер, но теперь мы вынуждены вводить значения в двоичном формате. Так что первым делом следует избавиться от пульта управления.

Конечно, его нужно заменить клавиатурой. Собранная ранее клавиатура прерывала работу микропроцессора при нажатии клавиши. Контроллер прерываний, который мы использовали, заставляет микропроцессор ответить на прерывание путем исполнения команды RST (Restart, «Перезагрузка»). Предположим, это команда RST1, и она заставляет микропроцессор сохранить текущее значение счетчика команд в стеке, а затем перейти по адресу 0008h. Начиная с этой ячейки, используя пульт управления, вы вводите некоторый код, который будем называть *обработчиком клавиатуры*.

Чтобы все работало правильно, понадобится код, который будет выполняться при перезагрузке микропроцессора, — код *инициализации*. Этот код сначала устанавливает указатель стека так, чтобы тот находился в допустимой области памяти, затем записывает во все ячейки видеопамяти шестнадцатеричное значение 20h, которое соответствует пробелу в кодировке ASCII. Это очищает экран от всех случайных символов.

Код инициализации использует команду OUT (Output, «Вывод») для установки курсора — символа подчеркивания, который обозначает место ввода следующего символа, — в первый столбец первой строки. Следующая команда EI разрешает обслуживание прерываний, благодаря чему микропроцессор может реагировать на сигналы прерывания клавиатуры. За ней следует команда HLT, останавливающая работу микропроцессора.

С кодом инициализации все готово. С этого момента компьютер в основном будет находиться в состоянии останова в результате выполнения инструкции HLT. Единственное событие, которое может вывести компьютер из этого состояния, — замыкание переключателя «Сброс» на пульте управления или прерывание от клавиатуры.

Код обработчика намного длиннее кода инициализации. Именно он отвечает за выполнение всех по-настоящему полезных функций.

При каждом нажатии клавиши на клавиатуре сигнал прерывания заставляет микропроцессор перейти от команды HLT в конце кода инициализации к обработчику клавиатуры, использующему инструкцию IN (Input, «Ввод») для определения нажатой клавиши, а затем производит некое действие в зависимости от нажатой клавиши (то есть *обрабатывает* ее нажатие) и выполняет

команду RET (Return, «Возврат») для возвращения процессора к команде HLT, где он будет ожидать следующего прерывания.

Если нажатой клавише соответствует буква, цифра или знак препинания, обработчик клавиатуры использует скан-код, чтобы определить соответствующий код ASCII с учетом того, была ли нажата клавиша Shift. Впоследствии он записывает этот код ASCII в видеопамять в соответствии с позицией курсора. Эта процедура называется *эхо* — отображение на экране символов, соответствующих нажатым клавишам. Курсор сдвигается на одну позицию вправо, располагаясь сразу после только что выведенного на экран символа. Таким образом в клавиатуру можно ввести множество символов, и они будут отображены на экране.

Если была нажата клавиша Backspace (соответствующая коду ASCII 08h), обработчик клавиатуры стирает последний записанный в видеопамять символ и перемещает курсор на одну позицию назад. (Стирание символа происходит путем записи кода ASCII 20h (символ пробела) в конкретную ячейку памяти.)

Обычно человек, работающий с клавиатурой, набирает символьную строку, пользуясь клавишей Backspace для исправления ошибок, а затем нажимает клавишу Return, часто обозначаемую словом Enter. Точно так же, как нажатие клавиши Return на электрической пишущей машинке указывает на необходимость перехода к началу следующей строки, нажатие Enter означает, что пользователь закончил набирать строку текста.

Когда обработчик клавиатуры взаимодействует с клавишей Return, или Enter (которая соответствует коду ASCII 0Dh), строка текста в видеопамяти интерпретируется как *команда* компьютеру, указание выполнить некое действие. Обработчик клавиатуры включает *командный процессор*, который понимает, например, три команды: W, D и R.

Если строка текста начинается с буквы W, она соответствует команде «Записать» (Write) некоторые байты в память. Допустим, на экране введена следующая строка.

```
W 1020 35 4F 78 23 9B AC67
```

Эта команда сигнализирует командному процессору, что нужно записать шестнадцатеричные байты 35, 4F в адреса памяти начиная с 1020h. Для выполнения этого задания обработчик клавиатуры должен перевести коды ASCII в байты. Такая трансформация обратна той, что я продемонстрировал ранее.

Если строка текста начинается с буквы *D*, она соответствует команде «Отобразить» (Display) некоторые из содержащихся в памяти байтов. В ответ

на строку командный процессор отображает на экране 11 байт, хранящихся в ячейках памяти начиная с 1030h.

```
D1030
```

Я упоминаю *11 байт*, потому что именно столько уместится в строке шириной в 40 символов после адреса. Вы можете использовать команду «Отобразить» для просмотра содержимого памяти.

Если строка текста начинается с буквы *R*, то она соответствует команде *Run* («Запустить»). Эта команда выглядит так:

```
R1000
```

и означает: «Запустить программу, которая хранится в ячейках памяти начиная с адреса 1000h». Командный процессор сохраняет адрес 1000h в пару регистров HL, затем выполняет команду PCHL, которая загружает в счетчик команд содержимое регистров HL, при этом осуществляется переход к соответствующему адресу.

Создание работоспособного обработчика клавиатуры и командного процессора — шаг вперед. После этого вам больше не придется мучиться с пультом управления. Ввод данных с клавиатуры проще, быстрее и аккуратнее.

Конечно, все еще остается нерешенной проблема исчезновения написанного кода при отключении питания. По этой причине вы, вероятно, решите хранить весь этот новый код в постоянной памяти, или ПЗУ. В предыдущей главе мы обсуждали микросхему ПЗУ, содержащую все комбинации точек, необходимые для отображения на экране символов ASCII, и предположили, что эти данные были «зашиты» в нашу микросхему при ее производстве. Вы также можете программировать микросхемы ПЗУ дома. *Программируемые постоянные запоминающие устройства* (ППЗУ) можно запрограммировать только один раз. *Стираемые программируемые постоянные запоминающие устройства* (СППЗУ) можно программировать и перепрограммировать после полного удаления данных ультрафиолетовым излучением.

Как вы помните, мы оснастили платы RAM DIP-переключателем, который позволяет указать ее начальный адрес. Если вы работаете с системой 8080, то для одной из плат RAM начальным адресом будет 0000h. После подключения микросхемы ПЗУ она займет адрес 0000h, а плату RAM можно будет переключить.

Создание командного процессора — серьезная веха не только потому, что он позволяет быстрее записывать байты в память, но и потому, что компьютер становится *интерактивным*. Когда вы вводите символы с клавиатуры, компьютер реагирует и отображает их на экране.

После записи командного процессора в ПЗУ вы можете начать экспериментировать с записью данных из памяти на жесткий диск (вероятно, фрагментами, размер которых соответствует размеру сектора диска) и считыванием этих данных обратно в память. Хранить программы и данные на диске намного безопаснее, чем в оперативной памяти, где они исчезнут в случае отключения питания, и гораздо удобнее, чем в ПЗУ.

Рано или поздно вам захочется добавить в командный процессор новое. Например, команда S означает Store («Сохранить»):

```
S2080 2 15 3
```

и указывает, что блок памяти, начинающийся с ячейки 2080h, должен быть сохранен на диске по адресу: сторона 2, дорожка 15, сектор 3. (Размер этого блока памяти зависит от размера сектора диска.) Аналогично вы можете добавить команду Load («Загрузить»), чтобы загрузить данные этого сектора с диска обратно в память.

```
L 2080 2 15 3
```

Разумеется, придется помнить, что и где вы храните. Для этого вы, вероятно, попробуете использовать блокнот и карандаш. Имейте в виду: нельзя просто сохранить код в ячейке памяти с определенным адресом, а затем загрузить его в ячейку памяти с другим адресом и ожидать, что он будет работать. Все команды Jump («Переход») и Call («Вызов») окажутся неправильными, поскольку в них записаны старые адреса. Кроме того, объем кода вашей программы может превысить размер сектора диска, поэтому придется хранить его в нескольких. Поскольку одни сектора на диске могут быть заняты другими программами или данными, а вторые быть свободными, сектора, в которых вы храните длинную программу, могут располагаться на диске непоследовательно.

Рано или поздно вы решите, что вручную отслеживать содержимое секторов диска слишком утомительно. Это будет означать, что вы дозрели до создания *файловой системы*.

Файловая система — способ хранения данных, при котором они организуются в *файлы*. Файл — это просто набор связанных данных, который занимает один или несколько секторов на диске. Самое главное, что у каждого файла собственное *имя*, помогающее определить его содержимое. Диск похож на шкаф для хранения документов, где каждая папка имеет этикетку с названием.

Файловая система почти всегда входит в состав более крупной совокупности программного обеспечения — *операционной системы*. Обработчик

клавиатуры и командный процессор, которые мы написали в этой главе, безусловно, могли бы эволюционировать в операционную систему. Однако вместо моделирования этого длительного процесса рассмотрим настоящую операционную систему (ОС) и постараемся разобраться, что она делает, как работает.

Исторически наиболее важной операционной системой для 8-битных микропроцессоров была CP/M (Control Program for Micros, управляющая программа для микрокомпьютеров), написанная в середине 1970-х для Intel 8080 Гэри Килдаллом (1942–1994), который позднее основал компанию Digital Research Incorporated (DRI).

На диске хранилась CP/M. На заре использования этой ОС наиболее распространенным носителем для нее была односторонняя 8-дюймовая дискета с 77 дорожками, 26 секторами на дорожке и 128 байтами в секторе (всего 256 байт). На первых двух дорожках диска содержалась сама CP/M. Чуть позже расскажу о том, как CP/M с диска переместилась в память компьютера.

Остальные 75 дорожек на диске CP/M использовались для хранения файлов. Файловая система CP/M довольно проста. Она удовлетворяет двум основным требованиям. Во-первых, каждый файл на диске имеет имя, которое тоже хранится на диске. На самом деле вся информация, которая требуется CP/M для считывания этих файлов, хранится на диске вместе с самими файлами. Во-вторых, файлы не обязательно должны занимать последовательные сектора. Часто при создании и удалении файлов разного размера свободное место на диске становится фрагментированным. Так что способность файловой системы хранить большой объем информации в несмежных секторах оказывается полезной.

Сектора на 75 дорожках, используемые для хранения файлов, сгруппированы в *блоки выделения памяти*. Каждый блок содержит восемь секторов, или 1024 байта. Всего на диске 243 блока выделения памяти, пронумерованных от 0 до 242.

Первые два блока выделения памяти (всего 2048 байт) отведены под *каталог*. Каталог — область диска, содержащая имена и некоторую важную информацию о каждом хранящемся файле. Каждому файлу на диске соответствует *элемент каталога* длиной 32 байта. Поскольку общий объем каталога всего 2048 байт, дискета может хранить 64 файла (2048 / 32).

Каждая 32-байтная запись каталога содержит следующую информацию.

Байты	**Значение**
0	Обычно равен 0
1–8	Имя файла
9–11	Тип файла
12	Экстент
13–14	Зарезервированы (равны 0)

Байты	Значение
15	Количество секторов в последнем блоке
16–31	Карта диска

Первый байт элемента каталога применяется только в том случае, когда с файловой системой работают два человека и более. В операционной системе CP/M этот байт обычно равен 0, как и байты тринадцатый и четырнадцатый.

В CP/M каждому файлу присваивается имя, состоящее из двух частей. Первая часть — *имя файла* — может содержать до восьми символов, хранящихся в байтах с первого по восьмой элемент каталога; вторая часть — *тип файла* — может содержать до трех символов, хранящихся в байтах с девятого по одиннадцатый. Существует несколько стандартных типов файлов. Например, TXT говорит о том, что файл является текстовым (содержит только коды ASCII), а COM (от command — «команда») указывает, что файл содержит машинные инструкции для процессора 8080, то есть программу. При указании конкретного файла эти две части разделяются точкой.

```
MYLETTER.TXT
CALC.COM
```

Этот метод наименования файлов известен как формат 8.3 («восемь точка три»), допускающий максимум восемь букв до точки и три буквы после.

Карта диска в элементе каталога указывает блоки выделения памяти, где хранится файл. Предположим, что значениями первых четырех байтов карты диска являются 14h, 15h, 07h и 23h, а значениями остальных — нули. Значит, файл занимает четыре блока, или четыре килобайта дискового пространства. Объем файла может быть немного меньше. Пятнадцатый байт в элементе каталога указывает, сколько 128-байтовых секторов фактически занято в последнем блоке.

Длина карты диска — 16 байт, и их достаточно для указания местоположения файла объемом до 16 384 байт. Для файла объемом более 16 килобайт необходимо использовать несколько элементов каталога, которые называются *экстентами*. В этом случае двенадцатый байт будет равен 0 в первом элементе каталога, 1 — во втором и т. д.

Ранее я упомянул текстовые файлы, которые также называются *ASCII-файлами* и содержат коды ASCII (включая коды возврата каретки и перевода строки), которые соответствуют понятным людям текстовым символам. Файл, который не является текстовым, называется *двоичным*. Файл CP/M типа COM — двоичный, поскольку содержит машинный код для процессора 8080.

Предположим, нам нужно сохранить в файле (очень маленького объема) три 16-битных числа, например 5A48h, 78BFh и F510h. Двоичный файл с этими тремя числами имеет длину всего шесть байт.

```
48 5A BF 78 10 F5
```

Разумеется, это формат хранения многобайтовых чисел для процессоров Intel. Первым указывается младший байт. Программа, написанная для процессоров Motorola, сохранила бы эти числа следующим образом.

```
5A 48 78 BF F5 10
```

В текстовом файле ASCII эти же четыре 16-битных значения были бы записаны, как показано ниже.

```
35 41 34 38 68 0D0A 37 38 42 46 68 0D0A 46 35 31 30 68 0D0A
```

Эти байты — коды ASCII для цифр и букв, а сами числа разделяются кодами возврата каретки (0Dh) и перевода строки (0A). Текстовый файл удобнее отображать не в виде строки кодов ASCII, а в виде соответствующих им символов.

```
5A48h
78BFh
F510h
```

Текстовый файл ASCII, в котором хранятся эти три числа, также может содержать следующие байты.

```
32 33 31 31 32 0D0A 33 30 39 31 31 0D0A 36 32 37 33 36 0D0A
```

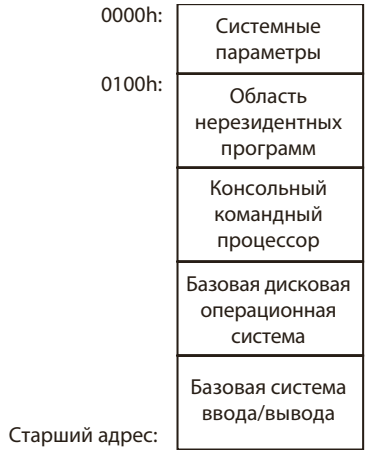
Коды ASCII для десятичных эквивалентов трех чисел следующие.

```
23112
30911
62736
```

Поскольку текстовые файлы призваны упростить людям процесс чтения их содержимого, нет причин для отказа от использования десятичных чисел вместо шестнадцатеричных.

Как я уже упоминал, сама CP/M записана на первых двух дорожках диска. Для запуска системы ее необходимо загрузить с диска в память. Объем ПЗУ в компьютере, использующем CP/M, не обязательно будет большим. В ПЗУ должен содержаться небольшой фрагмент кода, известный как *загрузчик программы*. Этот загрузчик считывает самый первый 128-байтовый сектор с дискеты в память и запускает его. Этот сектор содержит код для загрузки в память остальной части CP/M. Весь процесс называется *загрузкой* операционной системы.

В результате система CP/M размещается в оперативной памяти с самыми старшими адресами. После загрузки CP/M память будет организована так.



Эта схема не отражает реального масштаба. Три компонента CP/M: базовая система ввода/вывода (Basic Input/Output System, BIOS), базовая дисковая операционная система (Basic Disk Operating System, BDOS) и консольный командный процессор (Command and Control Processor, CCP) — занимают в общей сложности около шести килобайт памяти. Область нерезидентных программ — около 58 килобайт памяти на компьютере с оперативной памятью 64 килобайта — изначально не содержит ничего.

Консольный командный процессор эквивалентен командному процессору, созданному нами ранее. Словом, *консоль* обозначает совокупность клавиатуры и дисплея. Консольный процессор отображает на дисплее следующее *приглашение*.

```
A>
```

Приглашение — это сигнал, означающий возможность ввести некую команду. На компьютерах с несколькими дисками буква *A* указывает на первый диск,

с которого была загружена система CP/M. Вы вводите команды после приглашения и нажимаете клавишу Enter. Затем процессор CCP выполняет эти команды, в результате чего на экране обычно отображается некоторая информация. После завершения консольный процессор снова отобразит приглашение.

Процессор CCP распознает лишь несколько команд. Вероятно, наиболее важна следующая команда.

```
DIR
```

Эта команда отображает на экране содержимое каталога диска, то есть список всех хранящихся на диске файлов. Вы можете использовать специальные символы «?» и «*», чтобы отобразить только файлы с определенным именем или типом. Следующая команда, например, отображает текстовые файлы.

```
DIR *.TXT
```

Список всех файлов с именами из пяти символов, в которых первый символ — буква *A*, а последний — *B*, передает такая команда.

```
DIR A??? B.*
```

Еще одна важная команда — ERA (Erase, «Удалить»). Она используется для удаления файла с диска.

```
ERA MYLETTER.TXT
```

Следующая команда удаляет все текстовые файлы.

```
ERA *.TXT
```

Удаление файла означает освобождение соответствующего элемента каталога и дискового пространства.

Команда REN (Rename, «Переименовать») используется для изменения имени файла. Команда TYPE («Напечатать») показывает содержимое текстового файла. Поскольку текстовый файл содержит только ASCII-коды, эта команда позволяет читать файл прямо с экрана.

```
TYPE MYLETTER.TXT
```

Команда SAVE («Сохранить») сохраняет один или несколько 256-байтовых блоков памяти, расположенных в области временного хранения программ, в файл с указанным именем на диске.

Если вы введете команду, которую CP/M не сможет распознать, система будет интерпретировать данные в качестве имени программы, которая хранится на диске в виде файла. Файлы программ всегда имеют тип COM. Процессор CCP ищет файл с таким именем на диске. Если он существует, система CP/M загрузит файл с диска в область временного хранения программ, которая начинается с адреса памяти 0100h. Так запускаются программы, хранящиеся на диске. Например, если вы наберете

```
CALC
```

в ответ на приглашение CP/M, то если на диске присутствует файл с именем CALC.COM, процессор CCP загрузит этот файл в память начиная с адреса 0100h, а затем выполнит программу, перейдя к машинной инструкции, расположенной по адресу 0100h.

Ранее я объяснил, как можно вставить машинные инструкции в любую область памяти и выполнить их, однако в случае CP/M программы, которые хранятся в файлах на диске, должны предусматривать загрузку в память, начиная с определенного адреса — 0100h.

Система CP/M поставляется с несколькими полезными программами, включая PIP (Peripheral Interchange Program — программа взаимодействия с периферией), которая позволяет копировать файлы. Программа ED — текстовый редактор, который помогает создавать и изменять текстовые файлы. Программы вроде PIP и ED, имеющие небольшой объем и предназначенные для решения простых задач, часто называются служебными, или *утилитами*. Будучи пользователем системы CP/M, вы, вероятно, приобрели бы более крупные *программы-приложения*, например текстовые процессоры, электронные таблицы, или написали бы их самостоятельно. Все эти программы также хранятся в файлах типа COM.

Итак, мы разобрались с командами и утилитами CP/M, которые (как и в большинстве операционных систем) позволяют осуществлять элементарное управление файлами. Мы также поняли, как CP/M загружает программные файлы в память и выполняет их. Однако у операционной системы есть еще одна важная функция.

Программе, работающей под управлением CP/M, часто требуется вывести что-то на экран, считать то, что вы набрали на клавиатуре, прочитать файл с диска или записать его на диск. Однако в большинстве случаев программа

не записывает свои выходные данные непосредственно в видеопамять, не может получить доступ к аппаратному обеспечению клавиатуры, чтобы узнать, что вы набрали, и, конечно, не имеет доступа к оборудованию жесткого диска для чтения и записи его секторов.

Вместо этого программа, работающая под управлением CP/M, использует набор подпрограмм, встроенных в ОС для решения этих распространенных задач. Подобные подпрограммы были специально разработаны так, чтобы программы могли легко получить доступ ко всему оборудованию компьютера, включая монитор, клавиатуру и диск, не заставляя программистов беспокоиться, как эти периферийные устройства соединены. Главное, что программе, работающей под управлением CP/M, *необязательно* знать о дисковых секторах и дорожках. Это работа системы CP/M. Вместо этого она может сохранять целые файлы на диске, а затем считывать их.

Третья основная функция операционной системы — это предоставление программе легкого доступа к аппаратным средствам компьютера, то есть ее обеспечение *интерфейсом прикладного программирования* (Application Programming Interface, API).

Программа, работающая под управлением CP/M, использует API, сохраняя в регистре C определенное значение (называемое *номером функции*) и выполняя следующую команду.

```
CALL 5
```

Например, программа получает ASCII-код нажатой на клавиатуре клавиши путем выполнения представленных ниже команд.

```
MVI C, 01h
CALL 5
```

В результате ASCII-код нажатой клавиши будет содержаться в аккумуляторе A. Аналогично команды, указанные ниже, выводят на экран символ, соответствующий ASCII-коду, содержащемуся в аккумуляторе A, сдвигая курсор на одну позицию.

```
MVI C, 02h
CALL 5
```

Если программе требуется создать файл, она сохраняет в паре регистров DE адрес области памяти, которая содержит имя файла, а затем выполняет код.

```
MVI C, 16h
CALL 5
```

В данном случае в ответ на команду CALL 5 CP/M создает на диске пустой файл. Затем программа может использовать другие функции для того, чтобы записать в этот файл новые данные и *закрыть* его, или закончить работу. Эта же или другая программа может позднее *открыть* данный файл и прочитать его содержимое.

Что же делает команда CALL 5? В CP/M в ячейке памяти по адресу 0005h хранится инструкция JMP (Jump), которая осуществляет переход в область памяти, выделенной для базовой дисковой операционной системы (BDOS). В этой области содержится множество подпрограмм, которые выполняют каждую из функций CP/M. Базовая дисковая операционная система, как следует из ее названия, в первую очередь отвечает за поддержание работы файловой системы. Часто BDOS использует подпрограммы, находящиеся в базовой системе ввода/вывода (BIOS) — области, которая фактически обращается к аппаратным средствам клавиатуры, монитора и дисков. На самом деле BIOS — это единственный раздел системы CP/M, которому требуется информация об аппаратном обеспечении компьютера. Консольный командный процессор, а также служебные программы, поставляемые с CP/M, осуществляют работу, используя функции BDOS.

API является независимым от устройства интерфейсом, а это значит, что программе, написанной для системы CP/M, не обязательно «знать» механику работы клавиатуры или монитора конкретного компьютера, а также чтения и записи секторов диска. Она просто использует функции CP/M для решения задач, связанных с клавиатурой, монитором и диском. Приятное дополнение: программа CP/M способна работать на разных компьютерах, которые могут использовать различные аппаратные средства для доступа к этим периферийным устройствам. Тем не менее всем программам CP/M требуется микропроцессор Intel 8080 или процессор, понимающий инструкции 8080, например Intel 8085 или Zilog Z-80. Пока на компьютере работает CP/M, программа использует функции этой системы для опосредованного доступа к аппаратному обеспечению. Без стандартных API-интерфейсов программы нужно было бы специально подстраивать под разные типы компьютеров.

Когда-то CP/M была популярной операционной системой для микропроцессоров 8080, и ее важность для истории несомненна. Эта система существенно повлияла на 16-битную операционную систему под названием QDOS (Quick and Dirty Operating System — операционная система «на скорую руку»), написанную Тимом Патерсоном из Seattle Computer Products для 16-разрядных микропроцессоров 8086 и 8088 компании Intel. В итоге система QDOS была

переименована в 86-DOS и куплена корпорацией Microsoft. MS-DOS (Microsoft Disk Operating System — дисковая операционная система Microsoft) устанавливалась на первых персональных компьютерах IBM, выпущенных в 1981 году. Несмотря на то что 16-разрядная версия CP/M (под названием CP/M-86) также была доступна для IBM PC, стандартом быстро стала именно система MS-DOS. Лицензии на установку MS-DOS (которая называлась PC-DOS на компьютерах IBM) также продавались другим производителям IBM-совместимых компьютеров.

MS-DOS не унаследовала файловую систему CP/M, а использовала схему, названную *таблицей размещения файлов* (File Allocation Table, FAT), которую изобрели в Microsoft в 1977 году. Суть этой схемы в том, что дисковое пространство разделено на кластеры, размер которых в зависимости от емкости диска может варьироваться от 512 до 16 384 байт. Каждый файл — это набор кластеров. В соответствующем файлу элементе каталога указывается только его начальный кластер. В самой таблице FAT для каждого кластера на диске указывается, где находится продолжение файла.

Элементы каталога на диске MS-DOS имеют длину 32 байта и используют ту же систему наименования файлов 8.3, что и CP/M. Однако терминология здесь несколько иная: последние три буквы называются *расширением*, а не типом файла. В элементе каталога MS-DOS нет списка блоков выделения памяти. Вместо этого каталог содержит такую полезную информацию, как дата и время последнего изменения файла, а также его размер.

Ранние версии MS-DOS структурно напоминали CP/M. Однако система MS-DOS не нуждалась в BIOS, поскольку она уже содержалась в ПЗУ компьютера. Командный процессор MS-DOS — это файл с именем COMMAND.COM. Программы MS-DOS выпускаются в двух вариантах. Размер программ с расширением COM ограничен 64 килобайтами, более крупные имеют расширение EXE.

Несмотря на то что изначально система MS-DOS поддерживала интерфейс CALL 5 для функций API, для новых программ был рекомендован обновленный интерфейс, использующий механизм *программных прерываний*, который напоминает вызов подпрограммы, за исключением того, что этой программе необязательно знать фактический адрес, к которому она обращается. Программа вызывает функцию API-интерфейса MS-DOS, выполняя команду INT 21h.

Теоретически прикладные программы должны получать доступ к аппаратному обеспечению компьютера только через интерфейсы, предоставляемые операционной системой. Однако многие программисты, которые создавали прикладные программы для небольших компьютеров 1970-х годов и начала 1980-х, часто обходили эту операционную систему, особенно в плане работы с дисплеем. Программы, которые непосредственно записывали байты в видеопамять, выполнялись быстрее, чем программы, которые этого не делали. Действительно,

для некоторых приложений, особенно для тех, которым необходимо отображать на экране графику, эта операционная система была совершенно неприемлемой. В MS-DOS многим программистам больше всего нравилось, что она «не путалась под ногами» и давала возможность писать настолько быстрые программы, насколько позволяло аппаратное обеспечение.

По этой причине популярное программное обеспечение, работающее на компьютере IBM PC, часто опиралось на особенности его оборудования. Производители компьютеров, желавшие конкурировать с IBM, были вынуждены копировать эти особенности, в противном случае некоторые популярные программы на их компьютерах могли работать неэффективно или не работать вообще. Требования к оборудованию для таких программ часто включали пункт «IBM PC или 100%-совместимый компьютер».

В версию MS-DOS 2.0, выпущенную в марте 1983 года, была добавлена поддержка жестких дисков, в то время небольших (по сегодняшним меркам), однако их емкость стремительно увеличивалась. Чем больше емкость диска, тем больше файлов на нем можно хранить. А чем больше файлов может вместить диск, тем сложнее найти конкретный файл или использовать какой-либо метод их организации.

Решением этой проблемы в MS-DOS 2.0 стала *иерархическая файловая система*, которая была добавлена в уже существующую файловую систему с минимальным количеством изменений. Как вы помните, на диске есть область, называемая каталогом, который представляет собой список файлов и содержит информацию о том, где они хранятся. В иерархической файловой системе некоторые из этих файлов сами могут быть каталогами, то есть файлами, содержащими список других файлов. Некоторые из этих файлов также могут быть каталогами. Обычный каталог на диске называется *корневым каталогом*. Каталоги, содержащиеся в других каталогах, называются *подкаталогами*. Каталоги (иногда называемые папками) позволяют группировать связанные между собой файлы.

Иерархическая файловая система и некоторые другие функции MS-DOS 2.0 были позаимствованы из операционной системы UNIX, которая была разработана в начале 1970-х годов в Bell Telephone Laboratories. Ее основные авторы — Кен Томпсон (род. 1943) и Деннис Ритчи (1941–2011). UNIX изначально создавалась в качестве облегченной версии более ранней операционной системы Multics (Multiplexed Information and Computing Services), которую корпорация Bell Labs разрабатывала совместно с Массачусетским технологическим институтом и General Electric.

UNIX — излюбленная операционная система олдскульных программистов. В то время как бóльшая часть операционных систем создается для конкретных

компьютеров, UNIX разрабатывалась как *переносимая* операционная система, то есть способная адаптироваться под различные компьютеры.

Во времена разработки UNIX Bell Labs принадлежала компании American Telephone & Telegraph, поэтому на нее распространялись судебные постановления, призванные ограничить монопольное положение AT&T в телефонной отрасли. Первоначально AT&T было запрещено продавать UNIX; компания была вынуждена выдавать лицензии на ее использование другим организациям. Так что начиная с 1973 года такие лицензии были выданы многим университетам, корпорациям и правительственным организациям. В 1983 году AT&T наконец вернулась в компьютерный бизнес, выпустив собственную версию UNIX.

Именно поэтому единой модификации UNIX не существует. Есть различные версии под разными названиями, которые работают на различных компьютерах и продаются разными поставщиками. Многие люди внесли свой вклад в разработку UNIX. Те, кто добавляет новые элементы в данную систему, по-видимому, руководствуются преобладающей философией UNIX. Часть этой философии — широкое использование текстовых файлов. Многие служебные программы UNIX читают текстовые файлы, что-то с ними делают, а затем создают другой текстовый файл. Утилиты UNIX можно объединить в цепочки для выполнения разнообразных действий над этими текстовыми файлами.

UNIX изначально писалась для компьютеров, которые были слишком большими и дорогими для одного пользователя. С такими компьютерами могут одновременно взаимодействовать несколько пользователей благодаря технологии под названием «*разделение времени*». Ее суть следующая: к компьютеру подключается несколько дисплеев с клавиатурами, которые называются *терминалами*. Быстро переключаясь между всеми терминалами, операционная система может создать впечатление, будто компьютер одновременно обслуживает всех пользователей.

Операционная система, которая способна одновременно выполнять несколько программ, называется *многозадачной*. Очевидно, что сложность такой ОС значительно превышает сложность однозадачных CP/M и MS-DOS. Многозадачность усложняет файловую систему, поскольку несколько пользователей могут попытаться одновременно использовать одни и те же файлы. Кроме того, многозадачность влияет на то, как компьютер выделяет память различным программам, поэтому такой операционной системе требуется механизм *управления памятью*. Поскольку нескольким работающим одновременно программам необходимо больше памяти, существует вероятность, что у компьютера возникнет ее дефицит. Для решения этой проблемы в операционной системе можно реализовать технологию под названием *виртуальная память*: ненужные в данный момент блоки хранятся во временных файлах на диске, а затем

возвращаются в память, когда в них возникает необходимость (для этого используется так называемый файл подкачки, хранящийся на жестком диске).

Наиболее интересными явлениями в мире UNIX в последние годы стали Фонд свободного программного обеспечения (Free Software Foundation, FSF) и проект GNU, основанные Ричардом Столлманом. Аббревиатура GNU является акронимом выражения GNU's Not UNIX («GNU не UNIX»), что, разумеется, правда. Вместо этого программы, разработанные в рамках проекта GNU, являются совместимыми с операционной системой UNIX, однако распространяются так, что не могут стать чьей-либо собственностью. Благодаря проекту GNU появилось множество UNIX-совместимых служебных программ и инструментов, а также Linux — ядро операционной системы, совместимой с UNIX. Главный создатель ставшего в последние годы популярным ядра Linux — Линус Торвальдс, американский программист из Финляндии.

Однако главная тенденция в развитии ОС, зародившаяся в середине 1980-х, — это разработка таких крупных и сложных систем, как Apple Macintosh и Microsoft Windows, где графические пользовательские интерфейсы и широкие возможности видеосистем применяются для упрощения работы с приложениями. Эту тенденцию я опишу в глав 25.

Глава 23

Фиксированная точка, плавающая точка

В повседневной жизни мы легко оперируем целыми числами, дробями и процентами одновременно. Мы покупаем полдесятка яиц, заплатив налог в размере 8¼ процента из денег, полученных за 2¾ часа сверхурочной работы, оплаченной по тарифу, в полтора раза превышающему обычный. Большинство людей не испытывают трудностей при использовании таких величин. Услышав от статистиков о том, что «среднее американское домохозяйство состоит из 2,6 человека», мы не ужасаемся при мысли о связанных с этим повсеместных увечьях.

Тем не менее когда дело касается компьютерной памяти, переключение между целыми и дробными числами оказывается сложнее. Да, все данные хранятся в компьютерах в виде битов, то есть в виде двоичных чисел. Однако одни виды чисел выразить в битах гораздо легче, чем другие.

Сначала мы использовали биты для представления *положительных целых* или *положительных натуральных* чисел. Мы также узнали, как с помощью дополнения до двух можно отобразить *отрицательные целые* числа, чтобы упростить операцию сложения. В следующей таблице показано, какие диапазоны положительных и отрицательных целых чисел (отрицательные числа выражены с помощью дополнения до двух) можно хранить в ячейках памяти емкостью 8, 16 и 32 бит.

Число битов	Диапазон целых положительных чисел	Диапазон целых отрицательных чисел
8	От 0 до 255	От –128 до 127
16	От 0 до 65 535	От –32 768 до 32 767
32	От 0 до 4 294 967 295	От –2 147 483 648 до 2 147 483 647

Однако на этом мы и остановились. Помимо целых чисел математики также различают *рациональные* числа, которые могут быть представлены в качестве *отношения* двух целых чисел. Это отношение также называется *дробью*.

Например, дробь ¾ — рациональное число, отношение чисел 3 и 4. Это число также можно записать в виде *десятичной дроби*: 0,75. Десятичная дробь остается отношением двух чисел, в данном случае 75/100.

В главе 7 рассказывалось, что в десятичной системе счисления цифры слева от десятичного разделителя являются множителями целых *положительных* степеней числа 10, а цифры справа — множителями целых *отрицательных* степеней числа 10. В одном из примеров я показал, что число 42 705,684 равно:

$$
\begin{aligned}
&4 \times 10\,000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 : 10 + \\
&8 : 100 + \\
&4 : 1000.
\end{aligned}
$$

Обратите внимание на знаки деления. Затем я представил эту последовательность операций без деления:

$$
\begin{aligned}
&4 \times 10\,000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 \times 0{,}1 + \\
&8 \times 0{,}01 + \\
&4 \times 0{,}001.
\end{aligned}
$$

И наконец, отобразил это число, используя степени числа 10:

$$
\begin{aligned}
&4 \times 10^{4} + \\
&2 \times 10^{3} + \\
&7 \times 10^{2} + \\
&0 \times 10^{1} + \\
&5 \times 10^{0} + \\
&6 \times 10^{-1} + \\
&8 \times 10^{-2} + \\
&4 \times 10^{-3}.
\end{aligned}
$$

Существуют рациональные числа, которые не так легко представить в виде десятичной дроби, например ⅓. Если вы разделите 1 на 3, то получите следующее.

0,33333333333333333333333333333333333333333333333333333333…

И так до бесконечности. Подобная дробь называется периодической и записывается 0,(3). Несмотря на то что запись числа ⅓ в виде десятичной дроби выглядит неуклюже, это число по-прежнему является рациональным, поскольку это отношение двух целых чисел. Вот еще один пример: отношение $^1/_7$ можно записать в виде десятичной дроби.

0,142857142857142857142857142857142857142857142857142857…

Или $0,\overline{142857}$.

Иррациональные числа — это числа, которые нельзя выразить в виде отношения двух целых чисел, значит, десятичная дробь продолжается бесконечно без повторяющихся последовательностей цифр. К числам такого типа относится квадратный корень из 2.

$\sqrt{2} \approx 1,41421356237309504880168872420969807856967187537695…$

Квадратный корень из 2 — решение следующего алгебраического уравнения:

$$x^2 - 2 = 0.$$

Если число *не является* решением какого-либо алгебраического уравнения с целочисленными коэффициентами, оно называется *трансцендентным*. (Все трансцендентные числа иррациональны, однако не все иррациональные числа трансцендентны.) К трансцендентным числам относится число π, представляющее отношение длины окружности к ее диаметру и приблизительно равное следующему.

3,14159265358979328462643383279502884197169399375 11…

Еще одним трансцендентным числом является *e*, к которому стремится выражение:

$$(1 + \frac{1}{n})^n$$

при *n*, стремящемся к бесконечности. Данное число приблизительно равно следующему.

2,718281828459045235360287471352662497757247093699 96…

Все числа, о которых мы говорили, то есть рациональные и иррациональные, называются *действительными,* или *вещественными*. Это обозначение отличает их от *мнимых* — квадратных корней из отрицательных чисел. *Комплексные* числа — это комбинации мнимых и вещественных чисел. Несмотря на свое название, мнимые числа *существуют* и используются, например при решении некоторых сложных задач по электронике.

Мы привыкли считать, что числовой ряд *непрерывен*. Если вы дадите два рациональных числа, я определю, какое число находится между ними. Для этого достаточно найти их среднее арифметическое. Однако цифровые компьютеры не могут работать с континуумами. Биты могут быть равны либо 0, либо 1, между которыми нет больше никаких значений. Так что цифровые компьютеры могут иметь дело только с *дискретными* значениями. Количество дискретных значений, которые вы можете представить, напрямую связано с количеством доступных битов. Например, в ячейках емкостью 32 бита можно хранить положительные целые числа в диапазоне от 0 до 4 294 967 295. При необходимости сохранить значение 4,5 придется пересмотреть этот подход и действовать иначе.

Можно ли представить дробные значения в двоичном формате? Да, можно. Вероятно, самый простой подход — использование двоично-десятичного кода (BCD). Как говорилось в главе 19, кодировка BCD позволяет записать десятичные числа в двоичном формате. Для кодирования каждой десятичной цифры (0, 1, 2, 3, 4, 5, 6, 7, 8 и 9) требуется четыре бита.

Десятичная цифра	Двоичное значение
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Формат BCD особенно полезен в компьютерных программах, которые работают с денежными суммами. Самые очевидные примеры — программы для банков и страховых компаний; многие дробные числа в них предусматривают не более двух знаков после десятичного разделителя.

Как правило, для хранения двух BCD-цифр достаточно одного байта. Такая система записи иногда называется *упакованным кодом BCD*. В такой кодировке не используется дополнение до двух. По этой причине в случае упакованного кода BCD для указания того, является ли число положительным или отрицательным, обычно требуется дополнительный бит, называемый *знаковым битом*. Поскольку для хранения BCD-значения удобно выделять целое число байтов, под бит знака обычно отводится четыре или восемь бит памяти.

Предположим, что сумма денег, которой должна оперировать ваша компьютерная программа, никогда не превысит ±10 миллионов долларов. Другими словами, вам требуются значения от –9 999 999,99 до 9 999 999,99. Можно выделить по пять байт памяти для каждой сохраняемой суммы в долларах. Например, число –4 325 120,25 можно представить посредством пяти байт.

```
00010100  00110010  01010001  00100000  00100101
```

В шестнадцатеричном формате это эквивалентно следующей записи.

```
14h       32h       51h       20h       25h
```

Обратите внимание: крайняя левая тетрада равна 1, то есть число является отрицательным. Это знаковый бит. Если бы число было положительным, то крайняя левая тетрада была бы равна 0. Для представления каждой цифры в числе требуется по четыре бита, а прочитать их можно непосредственно по шестнадцатеричным значениям, поскольку они совпадают с десятичными.

Для представления значений в диапазоне от –9 999 999,99 до 9 999 999,99 вам понадобится шесть байт: пять байт для десяти цифр и еще целый байт для знакового бита.

Такой формат записи дробных чисел называется записью *с фиксированной точкой*, поскольку после десятичного разделителя всегда следует определенное количество цифр, в нашем примере две. Важно: данные о положении этого разделителя не хранятся вместе с числом. Программам, работающим с числами в таком формате, необходимо сообщить, где находится этот разделитель. Вы можете создавать числа с любым количеством десятичных знаков, а также использовать их в одной и той же компьютерной программе. Однако

любая часть программы, выполняющая над числами арифметические операции, должна знать, где находится десятичный разделитель.

Формат с фиксированной точкой хорошо работает только в том случае, если вы уверены, что числа не превысят размеры выделенных под них ячеек памяти, что вам не потребуется увеличивать количество десятичных знаков. Использование этого формата совершенно неуместно в ситуациях, когда числа могут стать слишком большими или маленькими. Предположим, вам нужно зарезервировать область памяти для хранения расстояний. Проблема в том, что эти расстояния могут значительно варьироваться. Расстояние от Земли до Солнца составляет 150 000 000 000 метров, а радиус атома водорода — 0,00000000005 метра. Для хранения значений в формате с фиксированной точкой, принадлежащих этому диапазону, придется выделить 12 байт памяти.

Возможно, мы сможем придумать более удобный способ хранения таких чисел, если вспомним, что ученые и инженеры выражают числа с помощью системы, называемой *научной нотацией* (*экспоненциальная запись*).

Научная нотация особенно полезна для представления очень больших и очень маленьких чисел, поскольку предусматривает использование степени числа 10, следовательно, позволяет обойтись без длинных строк нулей. В научной нотации следующие числа записываются следующим образом.

490 000 000 000	$4{,}9 \times 10^{11}$
0,00000000026	$2{,}6 \times 10^{-10}$

В этих двух примерах числа 4,9 и 2,6 называются *дробной частью,* или *мантиссой* (хотя этот термин более уместен для логарифмов). Однако я буду придерживаться компьютерной терминологии, называя этот фрагмент научной нотации *значащей частью числа*.

Порядок — это степень, в которую возводится число 10. В первом примере порядок равен 11, во втором — –10. Порядок показывает, на сколько мест был сдвинут десятичный разделитель в значащей части числа.

Существует соглашение, по которому значащая часть числа должна принадлежать интервалу от 1 (включительно) до 10. Несмотря на то что следующие числа равны, первый вариант представления является предпочтительным:

$$4{,}9 \times 10^{11} = 49 \times 10^{10} = 490 \times 10^{9} = 0{,}49 \times 10^{12} = 0{,}049 \times 10^{13}.$$

Такая форма научной нотации иногда называется *нормализованной**.

* А также экспоненциальной записью или записью с мантиссой. *Прим. науч. ред.*

Обратите внимание: знак показателя степени говорит только о порядке числа, но не о том, является ли оно отрицательным или положительным. Вот как выражаются отрицательные числа в научной нотации:

$$-5{,}8125 \times 10^{7} \text{ соответствует } -58\,125\,000;$$
$$-5{,}8125 \times 10^{-7} \text{ соответствует } -0{,}00000058125.$$

В компьютерах вместо формата с фиксированной точкой используется *формат с плавающей точкой*, который идеально подходит для хранения малых и больших значений, поскольку основан на научной нотации. Однако применяемый в компьютерах формат с плавающей точкой подразумевает запись в научной нотации *двоичных* чисел, поэтому нам необходимо выяснить, как выглядят дробные числа в двоичном формате.

Все гораздо проще, чем может показаться. В десятичной записи числа цифры справа от десятичного разделителя соответствуют отрицательным степеням числа 10. В двоичной записи цифры справа от *двоичного разделителя* (который выглядит так же, как и десятичный) соответствуют отрицательным степеням числа 2. Давайте преобразуем двоичное число в десятичное.

101,1101	$1 \times 4 +$
	$0 \times 2 +$
	$1 \times 1 +$
	$1 \div 2 +$
	$1 : 4 +$
	$0 : 8 +$
	$1 : 16$

Операции деления можно заменить умножением на отрицательные степени числа 2:

$$1 \times 2^{2} +$$
$$0 \times 2^{1} +$$
$$1 \times 2^{0} +$$
$$1 \times 2^{-1} +$$
$$1 \times 2^{-2} +$$
$$0 \times 2^{-3} +$$
$$1 \times 2^{-4}.$$

Отрицательные степени числа 2 можно также рассчитать путем последовательного деления 1 на 2:

$$
\begin{aligned}
&1 \times 4 + \\
&0 \times 2 + \\
&1 \times 1 + \\
&1 \times 0{,}5 + \\
&1 \times 0{,}25 + \\
&0 \times 0{,}125 + \\
&1 \times 0{,}0625.
\end{aligned}
$$

В результате этого вычисления находим, что десятичный эквивалент двоичного числа 101,1101 равен 5,8125.

В десятичной научной нотации нормализованная значащая часть числа должна быть больше или равна 1, но меньше 10. Таким же образом в двоичной научной нотации нормализованная значащая часть числа должна быть больше либо равна 1, но меньше числа 10, которое соответствует 2 в десятичной системе счисления. Выразим число в двоичной научной нотации.

$$101{,}1101 \qquad 1{,}011101 \times 2^2$$

Интересное следствие этого правила: слева от двоичного разделителя в нормализованном двоичном числе с плавающей точкой может стоять только 1.

У большинства современных компьютеров и программ, использующих числа с плавающей точкой, применяется стандарт, введенный в 1985 году Институтом инженеров электротехники и электроники (Institute of Electrical and Electronics Engineers, IEEE) и признанный Американским национальным институтом стандартов (American National Standards Institute, ANSI), — ANSI/IEEE Std 754–1985, *стандарт IEEE для двоичной арифметики с плавающей точкой*. В кратком описании этого стандарта, занимающем всего 18 страниц, хорошо изложены основы кодирования двоичных чисел с плавающей точкой.

Стандарт IEEE предусматривает два основных формата: число *одинарной точности*, занимающее в памяти четыре байта, и число *двойной точности*, занимающее восемь байт.

Сначала рассмотрим число одинарной точности. Оно состоит из трех частей: один бит отводится для знака (0 используется для положительного числа, а 1 — для отрицательного), восемь бит — для порядка, а 23 бита — для дробной значащей части числа, в которой самый младший бит — крайний справа.

1 знаковый бит (***s***)	8 битов порядка (***e***)	23 бита дробной значащей части (***f***)

Итого 32 бита, или четыре байта. Поскольку в значащей части нормализованного двоичного числа с плавающей точкой слева от двоичного разделителя всегда стоит 1, соответствующий ей бит не включается при сохранении числа в формате IEEE. Сохраняется только 23-битная *дробная часть*. Несмотря на то что для хранения значащей части числа используется только 23 бита, считается, что *точность* равна 24 битам. Чуть позже мы разберемся в том, что это значит.

Значение 8-битного порядка находится в диапазоне от 0 до 255. Такой порядок называется *смещенным*. Для нахождения истинного значения порядка с учетом знака необходимо вычесть из него число, называемое *смещением*. Для чисел одинарной точности с плавающей точкой смещение порядка равно 127.

Значения порядка 0 и 255 используются в особых случаях, о которых расскажу чуть позже. Если значение порядка принадлежит диапазону от 1 до 254, то число, представленное конкретными значениями *s* (бит знака), *e* (порядок) и *f* (дробная часть), равно:

$$(-1)^s \times 1,f \times 2^{e-127}.$$

Выражение $(-1)^s$ используется для определения знака числа. Если *s* равно 0, то число положительное (поскольку любое число в степени 0 равно 1), если *s* равно 1, то число отрицательное (поскольку –1 в степени 1 равно –1).

Следующая часть выражения 1,*f* представляет 1, за которой следует двоичный разделитель и 23-битная дробная значащая часть. Все это умножается на 2, возведенное в степень, значением которой является разность хранящегося в памяти 8-битного смещенного порядка и числа 127.

Я не упомянул о способе выражения такого распространенного числа, как 0. Похоже, о нем мы и забыли. Для этого предусмотрено несколько особых случаев:

— если *e* равно 0, *f* равно 0, то число равно 0; как правило, для представления числа 0 во все 32 бита записываются нули, однако бит знака может быть равен 1, и в этом случае число интерпретируется как *отрицательный ноль*; бит может обозначать очень маленькое отрицательное число, для представления которого с одинарной точностью доступных цифр и порядков недостаточно;

— если *e* равно 0, а *f* не равно 0, то число является действительным, но не нормализованным:

$$(-1)^s \times 0{,}f \times 2^{-127};$$

обратите внимание на 0 слева от двоичного разделителя значащей части;
— если *e* равно 255, а *f* равно 0, то число символизирует положительную или отрицательную бесконечность — в зависимости от знака *s*;
— если *e* равно 255, а *f* не равно 0, то значение считается «не числом» и обозначается аббревиатурой *NaN* (Not a Number — «не число»); NaN может указывать на неизвестное число или на результат недопустимой математической операции.

Наименьшее нормализованное положительное или отрицательное двоичное число, которое можно представить с одинарной точностью в формате с плавающей точкой, следующее:

$$1{,}00000000000000000000000_{\text{ДВА}} \times 2^{-126}.$$

В этом числе после двоичного разделителя следуют 23 двоичных нуля. Наибольшее нормализованное положительное или отрицательное двоичное число, которое можно представить с одинарной точностью в формате с плавающей точкой, следующее:

$$1{,}11111111111111111111111_{\text{ДВА}} \times 2^{127}.$$

В десятичной системе счисления эти два числа приблизительно равны $1{,}175494351 \times 10^{-38}$ и $3{,}402823466 \times 10^{38}$. Именно этими числами ограничивается диапазон чисел с плавающей точкой одинарной точности.

Вероятно, вы помните, что десять двоичных цифр примерно эквивалентны трем десятичным цифрам. Под этим подразумеваю, что двоичное число, состоящее из десяти единиц, которое соответствует числу 3FFh в шестнадцатеричном формате и 1023 в десятичном, приблизительно равно числу из трех девяток, то есть 999. Таким образом:

$$2^{10} \approx 10^3.$$

Из этого соотношения следует, что 24-битное двоичное число одинарной точности в формате с плавающей точкой приблизительно эквивалентно

десятичному числу, состоящему из семи цифр. По этой причине считается, что число одинарной точности в формате с плавающей точкой имеет *точность* до 24 битов, или около семи десятичных знаков. Что это значит?

Точность числа с фиксированной точкой очевидна. Например, денежная сумма, выраженная в виде числа с фиксированной точкой, имеющего два десятичных знака, определена с точностью до цента. Однако о числах с плавающей точкой мы не можем сказать ничего подобного. В зависимости от значения порядка число с плавающей точкой может иметь точность до долей цента или до десятков долларов.

Правильнее было бы сказать, что число одинарной точности с плавающей точкой имеет точность до одной части из 2^{24} (одной части из 16 777 216, примерно до шести частей из 100 миллионов). Что это значит *на самом деле*?

Во-первых, если вы попытаетесь выразить значения 16 777 216 и 16 777 217 в виде чисел одинарной точности с плавающей точкой, они окажутся одинаковыми. Более того, любое число в промежутке между этими двумя значениями (например, 16 777 216,5) тоже будет совпадать с ними. Все три десятичных числа сохраняются в памяти в виде 32-битного числа одинарной точности с плавающей точкой, которое, будучи разделенным на биты знака, порядка и значащей части, выглядит следующим образом.

4B800000h

0 10010111 00000000000000000000000

И оно эквивалентно

$$1{,}00000000000000000000000_{\text{ДВА}} \times 2^{24}.$$

Следующее значение, выраженное двоичным числом с плавающей точкой, эквивалентно числу 16 777 218:

$$1{,}00000000000000000000001_{\text{ДВА}} \times 2^{24}.$$

Хранение двух разных десятичных значений в виде одинаковых чисел с плавающей точкой не всегда создает проблемы.

Правда, если при написании банковской программы вы используете числа одинарной точности с плавающей точкой для хранения денежных сумм

в долларах и центах, вас, вероятно, будет беспокоить то, что 262 144,00 доллара равны 262 144,01 доллара. Обе эти суммы выражаются числом:

$$1{,}00000000000000000000000_{\text{ДВА}} \times 2^{18}.$$

Это одна из причин, почему при работе с долларами и центами предпочтительнее применять формат с фиксированной точкой. При использовании чисел с плавающей точкой вы можете обнаружить и другие раздражающие нюансы. Например, программа, выполняющая вычисление, в результате которого должно получиться число 3,50, выдает значение 3,499999999999. Так часто бывает при использовании чисел с плавающей точкой, и с этим ничего нельзя поделать.

Если вы твердо решили остановиться на числах с плавающей точкой, но вам недостаточно одинарной точности, попробуйте применить числа с плавающей точкой *двойной точности*. Числа в этом формате занимают восемь байт памяти, распределенных следующим образом.

1 знаковый бит (***s***)	11 бит порядка (***e***)	52 бита дробной значащей части (***f***)

Смещение порядка равно 1023, или 3FFh, поэтому число в этом формате записывается так:

$$(-1)^{s} \times 1{,}f \times 2^{e-1023}.$$

К нулю, бесконечности и значениям NaN применяются правила, аналогичные тем, которые мы рассматривали, когда говорили о числах одинарной точности.

Наименьшее положительное или отрицательное число двойной точности с плавающей точкой следующее:

$$1{,}00000000000000000000000000000000000000000000000000_{\text{ДВА}} \times 2^{-1022}.$$

В этом числе после двоичного разделителя следуют 52 нуля. Наибольшее число:

$$1{,}11111111111111111111111111111111111111111111111111_{\text{ДВА}} \times 2^{1023}.$$

Соответствующие десятичные числа формируют диапазон примерно от $2{,}2250738585072014 \times 10^{-308}$ до $1{,}7976931348623158 \times 10^{308}$. Число 10^{308} очень велико, оно представляет единицу с 308 нулями.

Пятидесятитрехбитная значащая часть числа (включая первый неучитываемый бит) приблизительно эквивалентна 16 десятичным знакам. Это уже намного лучше формата с одинарной точностью, однако вероятность того, что какое-то число однажды станет равно другому, по-прежнему существует. Возьмем, к примеру, числа 140 737 488 355 328,00 и 140 737 488 355 328,01. Они оба будут храниться в виде 64-битного числа двойной точности с плавающей точкой:

42E0000000000000h.

В двоичном формате это число выглядит так:

$$1{,}0000000000000000000000000000000000000000000000000000_{\text{ДВА}} \times 2^{47}.$$

Разумеется, разработка формата для хранения в памяти чисел с плавающей точкой является лишь небольшим этапом в процессе фактического использования этих чисел в программах, написанных на языке ассемблера. Если бы вы конструировали компьютер с нуля, сейчас бы возникла проблема создания набора функций для сложения, вычитания, умножения и деления чисел с плавающей точкой. К счастью, эти задачи можно разбить на более мелкие подзадачи, связанные со сложением, вычитанием, умножением и делением *целых* чисел, решать которые вы уже научились.

Например, сложение чисел с плавающей точкой сводится к сложению их значащих частей; сложность заключается лишь в порядках. Предположим, вам необходимо выполнить следующую операцию сложения:

$$(1{,}1101 \times 2^5) + (1{,}0010 \times 2^2).$$

В данном случае нужно сложить числа 11101 и 10010, однако второе число необходимо преобразовать с учетом разницы в значениях порядков. Фактически требуется сложить целые числа 11101000 и 10010. Итоговая сумма составит:

$$1{,}1111010 \times 2^5.$$

Иногда разница в порядках может быть такой большой, что одно из двух чисел даже не повлияет на сумму. Это может произойти, например, при сложении расстояния от Земли до Солнца и радиуса атома водорода.

Перемножение двух чисел с плавающей точкой сводится к перемножению двух значащих частей как обычных целых чисел и сложению двух целочисленных значений порядков. Нормализация значащей части может привести к уменьшению нового значения порядка.

Следующий уровень сложности при использовании чисел с плавающей точкой связан с вычислением квадратных корней, степеней, логарифмов и тригонометрических функций. Однако все эти задачи можно решить с помощью четырех основных операций: сложения, вычитания, умножения и деления.

Например, такая тригонометрическая функция, как синус, вычисляется с помощью разложения в ряд.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Аргумент *x* должен выражаться в *радианах*, где 360° — это 2π радиан, восклицательный знак — *факториал* числа, или произведение всех целых чисел от 1 и до этого числа. Например, 5! = 1 × 2 × 3 × 4 × 5. В данном случае все сводится к простому умножению. Возведение числителей дробей в степень тоже предполагает умножение. Остальными операциями являются деление, сложение и вычитание. Единственная по-настоящему сложная часть — многоточие в конце выражения, означающее, что это вычисление может продолжаться *бесконечно*. На практике если вы ограничитесь диапазоном от 0 до π/2 (из которого можно вывести все остальные значения синуса), то сможете избежать лишних вычислений. Вам достаточно десятка слагаемых в этом разложении для получения 53-битных значений двойной точности.

Если учесть, что компьютеры предназначены для того, чтобы облегчать людям жизнь, может показаться, что написание множества подпрограмм для выполнения арифметических операций с плавающей точкой противоречит цели их создания. Однако в этом вся прелесть программного обеспечения. Написанные кем-то подпрограммы для конкретного компьютера могут использоваться другими людьми. Арифметика с плавающей точкой настолько важна для научных и инженерных приложений, что ей традиционно придается огромное значение. На заре компьютерной эры при создании программного обеспечения для нового типа компьютеров написание подпрограмм для выполнения расчетов с плавающей точкой было одной из первоочередных задач.

Целесообразно разработать машинные инструкции специально для выполнения вычислений с плавающей точкой! Очевидно, это легче сказать, чем сделать, однако важность такой задачи трудно переоценить. Если вы сможете реализовать арифметику с плавающей точкой на уровне аппаратного обеспечения, подобно командам умножения и деления в 16-разрядных микропроцессорах,

то все вычисления с плавающей точкой будут выполняться компьютером гораздо быстрее.

Первым коммерческим компьютером, в котором вычисления с плавающей точкой могли осуществляться на аппаратном уровне, был IBM 704, выпущенный в 1954 году. Все числа в нем хранились в виде 36-битных значений. Числа с плавающей точкой разбивались на 27-битную значащую часть, 8-битный порядок и однобитный знак. Специальные аппаратные компоненты для расчетов с плавающей точкой могли выполнять операции сложения, вычитания, умножения и деления. Остальные функции реализовывались в программном обеспечении.

В настольном компьютере аппаратное обеспечение для вычислений с плавающей точкой появилось в 1980 году, когда компания Intel выпустила чип 8087. Интегральные микросхемы такого типа в наши дни называются *математическими сопроцессорами*, или *блоками вычислений с плавающей точкой*. Микросхема 8087 была названа *со*процессором, поскольку не могла работать сама по себе. Ее можно было использовать только в сочетании с первыми 16-разрядными микропроцессорами Intel 8086 и 8088.

Сопроцессор 8087 — микросхема с 40 выводами, которая использует многие из тех же сигналов, что и микропроцессоры 8086 и 8088. С помощью этих сигналов микропроцессор и математический сопроцессор взаимодействуют. Когда ЦПУ считывает специальную команду ESC (Escape), сопроцессор перехватывает управление и выполняет следующий машинный код, соответствующий одной из 68 команд для вычисления тригонометрических функций, степеней, логарифмов и т. д. Типы данных основаны на стандарте IEEE. В свое время сопроцессор 8087 считался самой сложной интегральной схемой.

Сопроцессор можно назвать небольшим автономным компьютером. В ответ на получение конкретной машинной инструкции для выполнения операции с плавающей точкой (например, команды FSQRT для вычисления квадратного корня) сопроцессор выполняет собственную последовательность команд, сохраненных в ПЗУ. Эти внутренние команды называются *микрокодом*. Как правило, эти вычисления осуществляются с помощью цикла, поэтому их результат предоставляется не сразу. Тем не менее математический сопроцессор обычно решает задачи по меньшей мере в десять раз быстрее, чем эквивалентные процедуры, реализованные в виде ПО.

Материнская плата первого компьютера IBM PC предусматривала 40-контактное гнездо для микросхемы 8087 рядом с процессором 8088. К сожалению, это гнездо было пустым. Пользователям, которым требовалось ускорить операции с плавающей точкой, приходилось приобретать микросхему 8087 отдельно и самостоятельно устанавливать ее. Однако даже после установки математического сопроцессора не все приложения начинали работать быстрее. Некоторые

программы, например текстовые редакторы, практически не нуждаются в вычислениях с плавающей точкой. Другие программы, вроде электронных таблиц, могут выполнять подобные вычисления гораздо чаще, поэтому они должны работать быстрее, однако от использования этой микросхемы скорость работы увеличивалась далеко не у всех приложений.

Дело в том, что программистам нужно было писать специальный код для использования машинных инструкций сопроцессора. Поскольку математический сопроцессор не являлся стандартным компонентом аппаратного обеспечения, многие себя этим не утруждали. В конце концов им все равно приходилось писать собственные подпрограммы для вычислений с плавающей точкой (поскольку у большинства пользователей не было математического сопроцессора), поэтому микросхема 8087 не освободила их от лишней работы, а наоборот, выявила дополнительные задачи. Со временем программисты научились писать приложения для использования математического сопроцессора, если он входил в состав компьютера, и эмулировать его работу, если не входил.

В дальнейшем компания Intel также выпустила математические сопроцессоры 287 и 387 для микропроцессоров 286 и 386 соответственно. В 1989 году появился процессор Intel 486DX, в который сопроцессор был встроен. С тех пор он перестал быть дополнительным компонентом. К сожалению, в 1991 году Intel сконструировала более дешевый микропроцессор 486SX без встроенного сопроцессора, а также отдельный математический сопроцессор 487SX. Однако с изготовлением процессора Pentium в 1993 году встроенный сопроцессор снова стал стандартом, вероятно навсегда. Компания Motorola интегрировала сопроцессор в микросхему 68040 в 1990 году. До этого Motorola продавала математические сопроцессоры 68881 и 68882 для изготовленных ранее микропроцессоров семейства 68000. Микросхемы PowerPC также предусматривают встроенный сопроцессор для выполнения вычислений с плавающей точкой.

Несмотря на то что аппаратный компонент для операций с плавающей точкой — подспорье для ассемблерного программиста, это устройство кажется малозначимой вехой в истории развития вычислительной техники, особенно если сравнивать с другими разработками, начатыми в 1950-х годах. Далее мы поговорим о языках программирования.

Глава 24
Языки высокого и низкого уровня

Писать программы в машинных кодах — все равно что есть с помощью зубочистки. Кусочки еды настолько малы, а процесс столь трудоемок, что обед может длиться вечно. Точно так же фрагменты машинного кода выполняют элементарные вычислительные задачи: загрузку числа из памяти в процессор, прибавление к нему другого числа, сохранение результата. На самом деле сложно понять, какую роль они играют.

По крайней мере, мы значительно продвинулись относительно описанного в начале главы 22 примитивного использования переключателей пульта управления для ввода в память двоичных данных. Тогда мы разобрались с написанием простых программ, позволяющих использовать клавиатуру и дисплей для ввода и просмотра шестнадцатеричных байтов машинного кода. Разумеется, это далеко не последнее из возможных улучшений.

Как вы знаете, байты машинного кода соответствуют некоторым коротким мнемоническим кодам, таким как MOV, ADD, CALL и HLT, благодаря чему компьютерные команды отдаленно напоминают английские слова. Эти мнемонические коды часто сопровождаются операндами, уточняющими действие машинной инструкции. Например, машинная инструкция для микропроцессора 8080 перемещает в регистр B содержимое ячейки памяти, 16-битный адрес которой хранится в паре регистров HL. Вот ее краткая запись.

```
MOV B, [HL]
```

Конечно, писать программы на языке ассемблера проще, чем в машинных кодах, однако микропроцессор этот язык не понимает. Я уже объяснял, как писать ассемблерные программы на бумаге. Когда вы будете готовы запустить подобную программу, вы вручную преобразуете инструкции на языке ассемблера в машинный код и введете их в память.

Прекрасно, если компьютер мог бы выполнить это преобразование вместо вас. Компьютер с процессором 8080, работающий под управлением операционной системы CP/M, предусматривает для этого все необходимые инструменты. Это работает следующим образом.

Сначала вы создаете текстовый файл, который будет содержать вашу программу, написанную на языке ассемблера. Для этого можете обратиться к программе CP/M ED.COM или к текстовому редактору, позволяющему создавать и изменять текстовые файлы. Предположим, вы создали текстовый файл с именем PROGRAM1.ASM. Тип ASM говорит, что файл содержит программу на языке ассемблера. Сам файл может выглядеть примерно так.

```
        ORG 0100h
        LXI DE, Text
        MVI C, 9
        CALL 5
        RET
Text:   DB 'Hello! $'
        END
```

В этом файле встречается пара новых для нас команд. Первая — ORG, которая *не является* частью системы команд процессора 8080; она указывает, что адрес следующей команды должен начинаться с ячейки 0100h. Как вы помните, именно с этого адреса CP/M загружает программы в память.

Следующая команда — LXI (Load Extended Immediate — непосредственная загрузка регистровой пары), загружающая 16-битное значение в пару регистров DE. В данном случае это 16-битное значение указывается в качестве метки Text в нижней части программы перед оператором DB (Data Byte — байт данных), который мы также встречаем впервые. За этим оператором могут следовать несколько байтов, разделенных запятыми (как в приведенном выше примере), или некоторый текст в одинарных кавычках.

Команда MVI (Move Immediate — передача непосредственного операнда) перемещает значение 9 в регистр C. Команда CALL 5 вызывает функции CP/M. Функция 9 отображает на экране строку символов, начинающуюся по адресу, который находится в паре регистров DE, и заканчивающуюся значком доллара. (То, что текст в последней строке программы завершается значком доллара, может показаться странным, однако именно так работает операционная система CP/M). Последняя команда, RET, завершает программу и возвращает управление системе CP/M. (На самом деле это лишь один из способов завершения программы CP/M.) Оператор END обозначает окончание файла на языке ассемблера.

Итак, у нас есть текстовый файл, содержащий семь строк текста. Теперь его нужно ассемблировать, то есть преобразовать в машинный код. Раньше мы делали это вручную. Однако теперь можем использовать предусмотренную в CP/M специально для этой цели программу-*ассемблер* ASM.COM. Она запускается из командной строки CP/M следующим образом.

```
ASM PROGRAM1.ASM
```

Программа ASM просматривает файл PROGRAM1.ASM и создает новый файл с именем PROGRAM1.COM, который содержит машинный код, соответствующий написанным нами на языке ассемблера командам. (Этот процесс включает еще один этап, однако он не имеет особого значения в описываемом примере.) Теперь вы можете запустить файл PROGRAM1.COM из командной строки CP/M. На экране отобразится текст Hello! — и все закончится.

Файл PROGRAM1.COM содержит следующие 16 байт.

```
11 09 01 OE09 CD05 00 C9 48 65 6C6C6F 21 24
```

Первые три байта — инструкция LXI, следующие два байта — инструкция MVI, следующие три байта — инструкция CALL, следующий байт — инструкция RET. Последние семь байт — это ASCII-коды, соответствующие пяти буквам слова Hello, восклицательному знаку и значку доллара.

Действия ассемблера, такого как ASM.COM, сводятся к чтению ассемблерной программы, часто называемой файлом с *исходным кодом*, и написанию *исполняемого* файла, содержащего машинный код. По большому счету ассемблеры довольно простые программы, поскольку между мнемоническими ассемблерными обозначениями команд и машинным кодом существует взаимно-однозначное соответствие. Ассемблер разделяет каждую текстовую строку на мнемокоды команд и аргументы, а затем сравнивает их с содержащимся в нем списком *всех* возможных мнемокодов и аргументов. Это сравнение показывает, какие машинные инструкции соответствуют каждой команде.

Обратите внимание на то, как ассемблер «понимает», что инструкция LXI должна сохранить в паре регистров DE адрес 0109h. Если сама инструкция LXI находится по адресу 0100h (а она там и находится, когда система CP/M загружает программу в память для последующего запуска), адрес 0109h соответствует началу текстовой строки. Обычно программисту, использующему ассемблер, не нужно беспокоиться о конкретных адресах, связанных с различными частями программы.

Создателю первого ассемблера, разумеется, пришлось вручную преобразовывать программу в машинный код. Человек, пишущий новый (возможно, улучшенный) ассемблер для того же компьютера, может воспользоваться языком ассемблера, чтобы затем преобразовать первый ассемблер. Как только новый ассемблер будет превращен в машинный код, он сможет ассемблироваться.

С выходом нового микропроцессора возникает необходимость в новом ассемблере. Однако новый ассемблер можно написать на существующем компьютере, используя его же транслятор. Когда ассемблер, работающий на компьютере А, создает код, который выполняется на компьютере Б, он называется кросс-ассемблером.

Несмотря на то что ассемблер избавляет от необходимости решать наименее творческие задачи программирования (вручную преобразовывать программу в машинный код), он не решает двух основных проблем, связанных с этим языком. Возможно, вы уже догадались, что первая проблема в том, что все действия с ассемблером могут быть крайне утомительными, поскольку вам приходится работать на уровне микропроцессора и беспокоиться о каждой мелочи.

Вторая проблема: написанные на языке ассемблера программы не являются *переносимыми*. Если вы пишете ассемблерную программу для микропроцессора Intel 8080, она не подойдет для микропроцессора Motorola 6800. Вам придется переписать ее на языке ассемблера 6800. Вероятно, это будет не так сложно по сравнению с написанием исходной программы, поскольку вы уже решили основные организационные и алгоритмические задачи. Однако это все равно потребует серьезных действий.

В предыдущей главе я говорил, что в современные микропроцессоры встроены машинные инструкции, выполняющие арифметические операции над числами с плавающей точкой. Это, безусловно, удобно, но это мало что дает. Предпочтительнее было бы полностью отказаться от машинно-зависимых инструкций, которые производят отдельные элементарные арифметические операции, а вместо этого выражать множество математических операций, используя проверенную временем алгебраическую форму записи. Например:

$$A \times \sin (2 \times \pi + B) / C,$$

где A, B и C — числа, а число π равно 3,14159.

Почему бы и нет? Если такое выражение записано в текстовом файле, у вас должна быть возможность написать ассемблерную программу, которая читает этот текстовый файл и преобразует алгебраическое выражение в машинный код.

Если бы вам требовалось вычислить значение этого алгебраического выражения только один раз, вы могли бы сделать это вручную или с помощью калькулятора. Вероятно, вы собираетесь использовать компьютер, поскольку вам необходимо вычислить значение этого выражения при многих различных значениях A, B и C. По этой причине вы также должны предусмотреть для данного алгебраического выражения некоторый контекст, позволяющий вычислять его значение при разных коэффициентах.

То, к созданию чего вы приблизились, называется *высокоуровневым* языком программирования. Язык ассемблера считается *низкоуровневым*, поскольку взаимодействует непосредственно с аппаратным обеспечением. Несмотря на то что термин «*язык высокого уровня*» используется для описания любого языка программирования, отличного от языка ассемблера, некоторые языки считаются более высокоуровневыми по сравнению с другими. Если бы вы, будучи президентом компании, могли бы сесть за компьютер и ввести команду (еще лучше просто положить ноги на стол и продиктовать): «Рассчитать все прибыли и убытки за этот год, написать годовой отчет, распечатать несколько тысяч копий и разослать всем нашим акционерам», это бы означало, что вы работаете с языком очень высокого уровня! В реальном мире языки программирования даже не приближаются к идеалу.

Человеческие языки — это сотни и тысячи лет сложных взаимодействий, случайных изменений и приспособлений. Даже в основе таких искусственных языков, как эсперанто, лежит реальный язык. Однако компьютерные языки высокого уровня — результат более целенаправленной работы. Задача изобретения языка программирования интересна для некоторых людей, поскольку язык определяет то, как человек передает инструкции компьютеру. По оценкам, сделанным в 1993 году, с начала 1950-х были изобретены и внедрены более тысячи языков высокого уровня.

Конечно, недостаточно просто *создать* высокоуровневый язык (что подразумевает разработку *синтаксиса*, то есть набора правил для составления выражений). Вы обязательно должны написать *компилятор* — программу, которая преобразует инструкции вашего высокоуровневого языка в машинный код. Подобно ассемблеру, компилятор должен прочитать файл с исходным кодом символ за символом и разбить его на короткие слова, символы и цифры. Тем не менее компилятор намного сложнее, чем ассемблер. Относительная простота последнего обусловлена взаимно-однозначным соответствием между инструкциями на языке ассемблера и машинным кодом. Компилятору обычно приходится преобразовывать одну инструкцию, написанную на языке высокого уровня, во множество машинных. Компиляторы писать нелегко, о чем свидетельствуют множество книг, посвященных их разработке.

Языки высокого уровня имеют свои преимущества и недостатки. Основная ценность в том, что их обычно легче изучать и использовать, чем языки ассемблера. Программы, написанные на языках высокого уровня, часто получаются более понятными и краткими. Кроме того, такие языки в основном являются переносимыми, то есть не зависят от конкретного процессора, что позволяет программисту не учитывать базовую структуру компьютера, где будет работать программа. Разумеется, если хотите запустить программу более чем на одном процессоре, вам потребуются компиляторы, которые генерируют машинный код для этих процессоров. Исполняемые файлы по-прежнему будут специфическими для каждого из них.

Однако код, написанный хорошим ассемблерным программистом, почти всегда превосходит код, созданный компилятором. Это означает, что исполняемый файл, получившийся из программы, написанной на языке высокого уровня, будет более объемным и медленным по сравнению с функционально идентичной программой, написанной на языке ассемблера. (В последние годы это стало менее очевидным в связи с усложнением микропроцессоров и усовершенствованием компиляторов в плане оптимизации кода.)

Несмотря на то что язык высокого уровня может упростить работу с процессором, он не сделает его более мощным. С помощью ассемблера вы можете получить доступ ко всем функциям процессора. Поскольку высокоуровневый язык необходимо преобразовывать в машинный код, он может только сократить возможности процессора. Действительно, если высокоуровневый язык является переносимым, он не может использовать функции, характерные для определенных процессоров.

Например, многие процессоры предусматривают команды побитового сдвига. Как вы помните, эти команды сдвигают содержащиеся в аккумуляторе биты вправо или влево. Однако таких команд практически не существует ни в одном высокоуровневом языке программирования*. Если перед вами стоит задача, требующая использования побитового сдвига, придется имитировать его путем умножения или деления на 2. (Нельзя сказать, что это плохо: многие современные компиляторы используют команды побитового сдвига процессора для умножения или деления на степень двойки.) Кроме того, во многих языках не предусмотрены и булевы операции над битами**.

На заре эры домашних компьютеров большинство прикладных программ были написаны на ассемблере. Однако в наши дни этот язык используется редко

---

* На сегодняшний день такие команды есть и в C++, и в C#, и в Python. Как правило, сдвиг влево и вправо задается конструкциями вида *x<<n, x>>n* соответственно, при этом *x* задает число, а *n* — количество разрядов, на которые нужно произвести сдвиг *x*. *Прим. науч. ред.*

** Тем не менее в наиболее распространенных языках высокого уровня такие операции имеются.

и только для решения особых задач*. По мере добавления в процессоры аппаратного обеспечения для конвейеризации — одновременного прогрессивного выполнения нескольких команд — язык ассемблера постоянно усложнялся. В то же время компиляторы становились все более интеллектуальными. Увеличение емкости диска и оперативной памяти современных компьютеров также сыграло свою роль: программистам больше не нужно создавать код, требующий небольшого объема памяти и умещающийся на небольшой дискете.

Несмотря на то что разработчики многих ранних компьютеров пытались формулировать для них задачи, используя алгебраическую форму записи, первым настоящим рабочим компилятором считается A-0 для компьютера UNIVAC, созданный в 1952 году Грейс Мюррей Хоппер (1906–1992) в корпорации Remington Rand. Доктор Хоппер пришла в компьютерную индустрию в 1944 году, когда присоединилась к команде Говарда Эйкена для работы над компьютером «Марк I». В свои восемьдесят с лишним лет она продолжала работать в этой сфере, занимаясь связями с общественностью в корпорации Digital Equipment Corporation (DEC).

Самый старый из используемых сегодня языков высокого уровня — ФОРТРАН (FORTRAN) (хотя за прошедшие годы он был многократно пересмотрен). Названия многих языков программирования пишутся прописными буквами, потому что являются акронимами. Название FORTRAN образовано от слов FORmula и TRANslation («трансляция формул»). Он был разработан в IBM для компьютеров серии 704 в середине 1950-х годов. На протяжении многих лет именно ФОРТРАНом пользовались ученые и инженеры. Он предусматривает обширную поддержку операций с плавающей точкой и работу с комплексными числами (которые, как я объяснил в предыдущей главе, представляют собой комбинации действительных и мнимых чисел).

У каждого языка программирования существуют сторонники и противники, которые могут горячо отстаивать свои предпочтения. В попытке занять нейтральную позицию при описании следующих концепций программирования я выбрал язык, который почти никто уже не использует, — АЛГОЛ (ALGOL — ALGOrithmic Language, алгоритмический язык; так же называется вторая по яркости звезда в созвездии Персея).

АЛГОЛ подходит для исследования природы высокоуровневых языков программирования, поскольку во многих отношениях это прямой предок многих популярных языков общего назначения, разработанных за последние 40 лет.

---

* Он также используется в ходе обучения программистов по вопросам, связанным с устройством и адресацией памяти персонального компьютера: на простых командах легче вникнуть в этот аспект. *Прим. науч. ред.*

Даже сегодня люди называют некоторые языки программирования языками типа АЛГОЛ.

Первая версия этого языка, АЛГОЛ 58, была разработана международным комитетом программистов в 1957 и 1958 годах. Два года спустя, в 1960-м, язык был доработан, а его пересмотренная версия получила название АЛГОЛ 60. Со временем появился АЛГОЛ 68, однако в этой главе я буду обращаться к версии, описанной в документе «Переработанное описание алгоритмического языка АЛГОЛ 60», который был завершен в 1962 году и впервые опубликован в 1963-м.

Давайте напишем краткий код на языке АЛГОЛ. Предположим, у нас есть компилятор под названием ALGOL.COM, который работает под управлением операционной системы CP/M или MS-DOS. Наша первая программа, написанная на языке АЛГОЛ, — текстовый файл с именем FIRST.ALG. Обратите внимание на тип файла ALG.

Программа на этом языке должна быть заключена между словами begin и end. Отобразим следующую текстовую строку.

```
begin
    print ('This is my fist ALGOL program!');
ende
```

Вы можете запустить компилятор, указав на программу FIRST.ALG, следующим образом.

```
ALGOL FIRST.ALG
```

Скорее всего, в ответ компилятор выведет что-то вроде этого.

```
Line 3: Unrecognized keyword 'ende'
```

К орфографическим ошибкам компилятор относится строже, чем придирчивый преподаватель. Я допустил ошибку в слове end, когда писал код, поэтому компилятор сообщил мне о наличии *синтаксической ошибки*. Вместо ende он ожидал встретить *ключевое слово*, которое способен распознать.

После исправления ошибки вы можете снова запустить компилятор. Иногда он может создать исполняемый файл напрямую (с именем FIRST.COM или FIRST.EXE в MS-DOS); иногда вам нужно выполнить еще одно действие. В любом случае, вскоре вы сможете запустить программу FIRST из командной строки.

```
FIRST
```

Программа FIRST выведет на экран следующую строку.

```
This is my fist ALGOL program!
```

Ой! Еще одна орфографическая ошибка (в слове first («первая») пропущена буква *r*). Компилятор *не в состоянии* обнаружить эту ошибку, поэтому она называется *ошибкой времени выполнения* — проявляется только при запуске программы.

Вероятно, вы уже поняли, что оператор print в нашей первой программе на языке АЛГОЛ отображает что-то на экране, в данном случае строку текста (эквивалент программы, написанной на языке ассемблера CP/M). На самом деле оператор print — это не часть официальной спецификации языка АЛГОЛ, но я предполагаю, что конкретный используемый нами компилятор предусматривает такую функцию, иногда называемую *встроенной*. Большинство операторов АЛГОЛ (не считая begin и end) должны сопровождаться точкой с запятой. Отступ для оператора print не обязателен, однако он часто применяется, чтобы сделать структуру программы более четкой.

Предположим, что вы хотите написать программу, которая перемножает два числа. В каждом языке программирования существует понятие *переменной*. Именем переменной в программе может быть буква, короткая последовательность букв или даже короткое слово. Переменная соответствует области памяти, однако в программе для обращения к ней используется имя, а не адрес памяти в числовом выражении. В этой программе задействованы три переменных с именами *a*, *b* и *c*.

```
begin
    real a, b, c;

    a:= 535.43;
    b:= 289.771;
    c:= a × b;

    print ('Произведение ', a, ' и ', b, ' равно ', c);
end
```

Оператор real необходим для *объявления* переменных в программе. В данном случае переменные называются *a*, *b* и *c* и представляют собой действительные числа или числа с плавающей точкой. (Для объявления целочисленных переменных в языке АЛГОЛ есть ключевое слово integer.) Как правило,

языки программирования требуют того, чтобы имена переменных начинались с буквы. Еще они могут содержать числа, но в переменных не должны употребляться пробелы и бóльшая часть других символов. Часто компиляторы ограничивают длину имени переменной. В приведенном в этой главе примере я использую просто буквы.

Если наш компилятор АЛГОЛ поддерживает стандарт IEEE для представления чисел с плавающей точкой, то для хранения каждой из трех переменных нужны четыре байта памяти (для чисел одинарной точности) или восемь байт памяти (для чисел двойной точности).

Следующие три выражения — операторы *присваивания*. В языке АЛГОЛ такой оператор легко узнается по двоеточию и знаку равенства. (В большинстве компьютерных языков в качестве оператора присваивания используется только знак равенства.) В левой части находится переменная, в правой — выражение. Переменной присваивается число, полученное в результате вычисления. Первые два оператора присваивания указывают, что переменным *a* и *b* назначаются конкретные значения. Третий оператор присваивает переменной *c* значение произведения переменных *a* и *b*.

В настоящее время использование всем известного символа умножения «×» в языках программирования обычно не допускается, поскольку он отсутствует в наборах символов ASCII и EBCDIC. В большинстве языков операция умножения обозначается звездочкой «*». Хотя в АЛГОЛе для деления используется косая черта «/», этот язык также предполагает употребление символа «÷» для операции целочисленного деления, результат которого показывает, сколько раз делитель умещается в делимом. Кроме того, для возведения в степень в языке АЛГОЛ используется символ «↑», который также не входит в набор ASCII.

Для отображения данных на экране применяется оператор print. Выводимые текст и переменные разделяются запятыми. Отображение символов ASCII, вероятно, не является для оператора print трудной задачей, однако в данном случае функция также должна преобразовать в символы ASCII числа с плавающей точкой.

```
Произведение 535.43 и 289.771 равно 155152.08653
```

Затем программа завершает работу и возвращает управление операционной системе.

Если хотите перемножить еще несколько чисел, нужно отредактировать программу, изменить числа, перекомпилировать и снова запустить ее. Вы можете избежать этой многократной перекомпиляции, воспользовавшись другой встроенной функцией под названием read.

```
begin
     real a, b, c;

     print ('Введите первое число: ');
     read (a);

     print ('Введите второе число: ');
     read (b);

     c:= a × b;

     print ('Произведение ', a, ' и ', b, ' равно ', c);
end
```

Оператор read считывает символы ASCII, которые вы вводите с клавиатуры, и преобразует их в числа с плавающей точкой.

В языках высокого уровня важной конструкцией является *цикл*, позволяющий написать программу, выполняющую одни и те же действия с разными значениями переменной. Предположим, вы хотите, чтобы программа вычисляла кубы чисел 3, 5, 7 и 9. Это можно сделать таким образом.

```
begin
     real a, b;

     for a:= 3, 5, 7, 9 do
     begin
          b:= a × a × a;
          print ('Куб числа ', a, ' равен ', b);
     end
end
```

Оператор for сначала присваивает переменной a значение 3, а затем выполняет команду, следующую за ключевым словом do. Если команд, которые требуется выполнить, несколько (как в приведенном примере), то между ключевыми словами begin и end необходимо задействовать несколько операторов. Эти два ключевых слова определяют *блок* операторов. Затем оператор for выполняет те же команды для переменной *a*, которой присвоены значения 5, 7 и 9.

Вот еще одна версия инструкции for, которая вычисляет кубы нечетных чисел от 3 до 99.

```
begin
    real a, b;

    for a:= 3 step 2 until 99 do
    begin
        b:= a × a × a;
        print ('Куб числа ', a, ' равен ', b);
    end
end
```

Сначала оператор for присваивает переменной a значение 3 и выполняет блок операторов, следующий за инструкцией for. Затем значение переменной *a* увеличивается на величину, следующую за ключевым словом step, то есть на 2. Новое значение переменной a (5) используется для выполнения блока операторов. Значение переменной a будет и дальше увеличиваться на 2. Когда оно превысит 99, цикл for завершится.

Как правило, языки программирования имеют строгий синтаксис. Например, в АЛГОЛе 60 за ключевым словом for может следовать только имя переменной. Однако в английском после слова for могут располагаться всевозможные слова. Несмотря на сложность написания компиляторов, создать их намного проще, чем программы для интерпретации человеческой речи.

Еще одна важная особенность большинства языков программирования — *условные структуры*, позволяющие выполнить некоторое действие только в случае истинности конкретного условия. Вот пример использования встроенной функции языка АЛГОЛ sqrt, которая вычисляет квадратный корень. Функция sqrt не работает с отрицательными числами, и данная программа это учитывает.

```
begin
    real a, b;

    print ('Введите число: ');
    read (a);

    if a < 0 then
        print('Извините, введено отрицательное число.');
    else
       begin
         b:= sqrt(a);
```

```
            print ('Квадратный корень из ', a, ' равен ', b);
    end
end
```

Левая угловая скобка (<) — знак «меньше». Если пользователь введет отрицательное число, то будет выполнен первый оператор print. При вводе положительного числа будет запущен блок операторов, содержащий другой print.

До сих пор каждая из переменных в приведенных выше программах хранила лишь одно значение. Часто бывает удобно хранить в одной переменной несколько значений. В этом случае переменная называется *массивом*. В программе на языке АЛГОЛ массив объявляется следующим образом.

```
real array a[1:100];
```

В данном случае мы указали, что хотим использовать эту переменную для хранения 100 различных чисел с плавающей точкой, называемых *элементами* массива. Для обращения к первому элементу массива используется выражение a[1], ко второму — a[2], к последнему — a[100]. Число в квадратных скобках называется *индексом* элемента массива.

Эта программа вычисляет квадратные корни всех чисел от 1 до 100 и сохраняет их в массиве. Затем она выводит их на экран.

```
begin
    real array a[1:100];
    integer i;

    for i:= 1 step 1 until 100 do
        a[i]:= sqrt(i);

    for i:= 1 step 1 until 100 do
        print ('Квадратный корень из ', i, ' равен ', a[i]);
end
```

Кроме того, программа показывает *целочисленную* переменную с именем *i*, которое является традиционным, поскольку это первая буква в слове integer («целое»). В первом цикле for каждому элементу массива присваивается значение квадратного корня из его индекса, во втором элементы массива выводятся на экран.

Помимо типов real и integer, АЛГОЛ предусматривает тип переменных Boolean. (Помните Джорджа Буля из главы 10?) Такая переменная может иметь только два возможных значения: true и false. Буду использовать массив булевых переменных (и почти все, что мы изучили до сих пор) в последней программе этой главы, в которой реализуется известный алгоритм нахождения простых чисел под названием «Решето Эратосфена». Эратосфен (около 276–196 до н. э.) служил главным библиотекарем в легендарной Александрийской библиотеке и сегодня широко известен благодаря вычислению точной длины окружности Земли.

Простыми являются целые числа, которые без остатка делятся только сами на себя и на 1. Первое простое число — 2 (единственное четное простое число), затем следуют 3, 5, 7, 11, 13, 17 и т. д.

Первый шаг в алгоритме Эратосфена — составление списка положительных целых чисел начиная с 2. Поскольку 2 — простое число, необходимо удалить все числа, кратные двум (все четные числа, кроме 2), а так как 3 — простое число, следует исключить все числа, кратные 3. Мы уже знаем, что 4 не является простым числом, потому что оно было вычеркнуто. Следующее простое число — 5, значит, нужно удалить все числа, кратные 5. Продолжая действовать таким способом, вы будете находить новые простые числа.

В программе для нахождения простых чисел от 2 до 10 000, написанной на языке АЛГОЛ, этот алгоритм можно реализовать, объявив булев массив с индексами от 2 до 10 000.

```
begin
    Boolean array a[2:10000];
    integer i, j;

    for i:= 2 step 1 until 10000 do
a[i]:= true;

    for i:= 2 step 1 until 100 do
        if a[i] then
              for j:= 2 step 1 until 10000 ÷ i do
                    a[i × j]:= false;

    for i:= 2 step 1 until 10000 do
        if a[i] then
              print (i);
end
```

Первый цикл for присваивает всем элементам булева массива значение true. Таким образом, предполагается, что все числа, находящиеся в начале программы, простые. Второй цикл проверяет числа от 1 до 100 (квадратный корень из 10 000). Если число простое (a[i] имеет значение true), то вложенный цикл for задает значение false всем числам, кратным ему. Последний цикл for выводит на экран все простые числа, то есть значения i, при которых a[i] — true.

Иногда люди спорят, является ли программирование искусством или наукой. С одной стороны, существуют учебные программы в области *компьютерных наук*, а с другой — есть такие знаменитые книги, как «*Искусство программирования*» Дональда Кнута. Как писал физик Ричард Фейнман: «Информатика скорее подобна инженерному делу: нужно просто заставить что-то делать что-то».

Если попросите 100 разных людей написать программу, которая выводит на экран простые числа, получите 100 различных решений. Даже те программисты, которые используют алгоритм Эратосфена, реализуют его не так, как это сделал я. Если бы программирование действительно было наукой, не существовало бы такого множества возможных решений, а неправильные заключения были бы более очевидными. Иногда стоящая перед программистом проблема вызывает у него озарения и творческие вспышки: в этом и заключается «искусство». И все же программирование в основном сводится к проектированию и строительству, подобно процессу возведения моста.

Многие из первых программистов были учеными и инженерами, способными формулировать задачи в виде математических алгоритмов, что было необходимо при использовании языков ФОРТРАН и АЛГОЛ. Тем не менее на протяжении истории развития этой сферы осуществлялись попытки создания языков для более широкого круга пользователей.

Одним из первых удачных языков, предназначенных для бизнеса, был КОБОЛ (COBOL, COmmon Business Oriented Language, «универсальный язык, ориентированный на коммерческие задачи»), который широко распространен и сегодня. Разработка языка КОБОЛ была начата в 1959 году комитетом, состоявшим из представителей промышленности и министерства обороны США, и большое влияние на него оказали ранние компиляторы Грейс Хоппер. Язык КОБОЛ был создан так, чтобы менеджеры, не занимавшиеся написанием кода, могли по крайней мере прочитать его и проверить, что он делает именно то, что от него ожидается. (Однако в реальной жизни так бывает нечасто.)

В языке КОБОЛ предусмотрены обширные возможности для чтения *записей* и создания *отчетов*. Запись — это набор взаимосвязанных данных. Например, страховая компания может хранить большие файлы с информацией обо всех проданных ею полисах. Каждому полису будет соответствовать отдельная пометка, включающая имя клиента, дату рождения и прочие данные. Многие

ранние программы на языке КОБОЛ были написаны для работы с 80-столбцовыми записями, хранящимися на перфокартах IBM. Для экономии места на этих картах годы часто кодировались с помощью двух цифр вместо четырех, что в дальнейшем послужило причиной широко распространенной «проблемы 2000 года».

В середине 1960-х IBM в связи со своим проектом System/360 разработала язык под названием ПЛ/1 (PL/I, Programming Language I, «язык программирования номер один»). Цель его создания — объединить блочную структуру АЛГОЛа, математические функции ФОРТРАНа и средства КОБОЛа для работы с записями. Однако этот язык так никогда и не достиг популярности ФОРТРАНа и КОБОЛа.

Несмотря на существование версий ФОРТРАНа, АЛГОЛа, КОБОЛа и ПЛ/1 для домашних компьютеров, главным языком для них стал БЕЙСИК.

Язык БЕЙСИК (BASIC, Beginner's All-purpose Symbolic Instruction Code, «универсальный код символических инструкций для начинающих») был придуман в 1964 году профессорами математического факультета Дартмутского университета Джоном Кемени и Томасом Курцем для работы с университетской системой распределения времени. Большинство студентов Дартмута не были математиками или инженерами, поэтому никто не ожидал, что они будут возиться с перфокартами и сложным синтаксисом. Вместо этого студент, сидя перед терминалом, мог написать простую программу, просто набрав команды, которым предшествовали числа, обозначающие их порядок. Команды, не сопровождающиеся числами, предназначались для системы: SAVE (сохранить на диске), LIST (отобразить строки по порядку), RUN (скомпилировать и запустить). Вот первая программа на языке БЕЙСИК, опубликованная в первом печатном руководстве.

```
10 LET X = (7 + 8) / 3
20 PRINT X
30 END
```

В отличие от АЛГОЛа язык БЕЙСИК не требовал указывать тип переменной. Большинство чисел сохранялись в виде значений с плавающей точкой.

Многие последующие реализации языка БЕЙСИК имели форму *интерпретаторов*, а не *компиляторов*. Как я объяснял, компилятор читает файл с исходным кодом и создает исполняемый файл. Интерпретатор читает исходный код и сразу выполняет его, не создавая исполняемого файла. По сравнению с компиляторами интерпретаторы легче писать, однако интерпретируемая программа выполняется медленнее, чем скомпилированная. На домашних компьютерах

язык БЕЙСИК начал использоваться в 1975 году, когда два приятеля, Билл Гейтс (род. 1955) и Пол Аллен (род. 1953), написали интерпретатор языка БЕЙСИК для микрокомпьютера Altair 8800 и основали корпорацию Microsoft.

Язык программирования Паскаль (Pascal), который унаследовал бóльшую часть своей структуры от АЛГОЛа, но включал функции обработки записей КОБОЛа, был разработан в конце 1960-х годов швейцарским профессором информатики Никлаусом Виртом (род. 1934). Язык Паскаль был довольно популярен среди программистов, работавших на компьютере IBM PC, но лишь в специфической реализации — Turbo Pascal, выпущенной компанией Borland International в 1983 году. Реализация Turbo Pascal, написанная датским студентом Андерсом Хейлсбергом (род. 1960), представляла версию, дополненную *интегрированной средой разработки* (Integrated Development Environment, IDE). Текстовый редактор и компилятор были объединены в одну программу, что значительно ускорило процесс программирования. Интегрированные среды разработки широко использовались на мейнфреймах, а среда Turbo Pascal ознаменовала начало их применения на небольших компьютерах.

Паскаль также серьезно повлиял на язык Ада (Ada), созданный для министерства обороны США. Язык был назван в честь Августы Ады Байрон, о которой я упоминал в главе 18, когда описывал аналитическую машину Чарльза Бэббиджа.

Затем появился очень популярный язык C (Си), созданием которого в период с 1969 по 1973 год в основном занимался Деннис Ритчи из Bell Telephone Laboratories. Часто спрашивают, почему язык называется C. Все просто: он был написан на основе более раннего языка B — упрощенной версии BCPL (Basic CPL), предшественником которого являлся CPL (Combined Programming Language, «комбинированный язык программирования»).

В главе 22 я упоминал, что операционная система UNIX задумывалась как переносимая. Большинство операционных систем в то время были разработаны на языке ассемблера для конкретного процессора. В 1973 году UNIX была написана (вернее, переписана) на языке C, и с тех пор эта система и этот язык тесно связаны.

Как правило, в языке C операторы записываются весьма кратко. Например, вместо ключевых слов begin и end, используемых в АЛГОЛе и Паскале для разграничения блоков, в C применяются фигурные скобки { и }. Вот еще один пример. Программисту часто требуется увеличить значение переменной на некую постоянную величину.

```
i = i + 5;
```

В C вы можете сократить этот оператор.

```
i += 5;
```

Если вам нужно увеличить значение переменной на 1 (инкрементировать ее), данный оператор можно записать еще короче.

```
i++;
```

Такой оператор на 16- или 32-разрядном микропроцессоре может выполняться с помощью одной машинной инструкции.

Ранее я упоминал, что бóльшая часть языков высокого уровня не предполагает операций побитового сдвига или булевых операций над битами, которые являются частью функционала многих процессоров. Язык C — исключение из правила. Кроме того, важная особенность этого языка — поддержка *указателей*, которые, по сути, являются числовыми представлениями адресов памяти. Поскольку язык C поддерживает операции, реализующие многие инструкции процессора, он иногда классифицируется как *язык ассемблера высокого уровня*. По сравнению с любым другим языком типа АЛГОЛ язык C точнее всего имитирует общие наборы команд процессора.

Тем не менее *все* языки типа АЛГОЛ, то есть *наиболее* распространенные языки программирования, разрабатывались для компьютеров с архитектурой фон Неймана. Выйти за рамки соответствующего образа мышления непросто, а заставить других людей использовать такой язык еще сложнее. Один из подобных языков — LISP (List Processing, «обработка списков»), созданный Джоном Маккарти в конце 1950-х годов и пригодный для работы в области искусственного интеллекта. Другой язык, столь же необычный, как LISP, но совершенно на него не похожий, — APL (A Programming Language, «язык программирования») — был придуман в конце 1950-х Кеннетом Айверсоном. В нем используется набор специальных символов, которые выполняют операции одновременно над целыми массивами чисел.

Несмотря на то что языки типа АЛГОЛ продолжают доминировать, в последние годы они были несколько усовершенствованы, что привело к появлению так называемых *объектно-ориентированных* языков, удобных при работе с графическими операционными системами, о которых я расскажу в следующей главе.

Глава 25
Графическая революция

В выпуске журнала Life от 10 сентября 1945 года читатели обнаружили привычную чересполосицу статей и фотографий: заметки об окончании Второй мировой войны, рассказ о жизни танцора Вацлава Нижинского в Вене, фоторепортаж о профсоюзе рабочих автомобильной промышленности. Однако тот выпуск содержал и нечто неожиданное: провокационную статью Вэнивара Буша (1890–1974) о будущем научных исследований. С 1927 по 1931 год, работая профессором Массачусетского технологического института, Ван Буш (как его называли) внес свой вклад в историю вычислительной техники, сконструировав один из самых значимых аналоговых компьютеров — дифференциальный анализатор. В 1945 году, когда вышел его артикул, Буш возглавлял управление научных исследований и разработок, которое во время войны координировало научные исследования США, включая Манхэттенский проект.

Статья Буша «Как мы можем мыслить» (As We May Think), несколько сокращенная с момента своего первого появления в The Atlantic Monthly двумя месяцами ранее, содержала описания некоторых будущих гипотетических изобретений, предназначенных для ученых и исследователей, которым приходится иметь дело с постоянно растущим объемом специализированных изданий. Решение этой проблемы Буш видел в использовании микропленки и придумал устройство под названием Memex для хранения книг, статей, звукозаписей и изображений. Это устройство также должно было позволять пользователю устанавливать тематические связи между этими фрагментами информации по аналогии с тем, как человеческий разум формирует ассоциации. Буш даже представлял новую группу профессионалов, которые должны были заниматься созданием этих ассоциативных связей.

Несмотря на то что на протяжении всего XX века о чудесах будущего было написано множество работ, статья Буша выделялась. В ней речь не шла о бытовых устройствах для облегчения домашнего труда, о футуристических видах транспорта или о роботах. Она была посвящена *информации* и тому, как новые технологии могут помочь справиться с увеличением ее объема.

За десятилетия, прошедшие с момента создания первых релейных калькуляторов, компьютеры стали меньше, быстрее и дешевле. Эта тенденция изменила саму природу вычислений. Чем более дешевыми становятся компьютеры, тем больше людей могут их себе позволить. По мере уменьшения их размера и повышения роста быстродействия программное обеспечение оказывается все более изощренным, благодаря чему расширяется круг задач, решаемых этими машинами.

Дополнительная мощность и скорость могут использоваться для совершенствования самой важной части компьютерной системы — *пользовательского интерфейса*, который обеспечивает человеко-машинные взаимодействия. Люди и компьютеры — разные создания. К сожалению, людей гораздо легче убедить в необходимости приспосабливаться к особенностям вычислительных машин, чем наоборот.

Поначалу цифровые компьютеры не были интерактивными. Для программирования одних использовались переключатели и кабели, для программирования других — перфорированная лента или пленка. В период с 1950-х до начала 1970-х годов компьютеры эволюционировали до такой степени, что нормой стала *пакетная обработка* с минимальным вмешательством оператора: программы и данные сохранялись на перфокартах, которые затем считывались в память. Программа анализировала полученную информацию, делала некоторые выводы и печатала результаты на бумаге.

Самые ранние интерактивные компьютеры работали с помощью телетайпных аппаратов. Такие установки, как система распределения времени Дартмутского университета (начала 1960-х), описанная в предыдущей главе, позволяли одновременно применять несколько телетайпов. Пользователь набирал на телетайпе строку текста, на которую компьютер отвечал одной или несколькими строками, отпечатываемыми на рулоне бумаги. Обмен информацией между телетайпом и компьютером осуществлялся с помощью потоков ASCII-кодов (хотя могла быть и другая кодировка), включающих, помимо кодов символов, простые управляющие коды, например для возврата каретки и перевода строки.

Однако электронно-лучевая трубка, которая получила более широкое распространение в 1970-е, не имела таких ограничений. Программное обеспечение могло использовать экран более гибко — в качестве двумерной платформы для отображения информации. Тем не менее разработчики большинства программ для небольших компьютеров, вероятно привыкшие к определенной логике отображения информации, продолжали рассматривать экран в качестве «стеклянного телетайпа». Создаваемые ими программы выводили данные на экран построчно сверху вниз, а по достижении нижней границы его

содержимое необходимо было прокручивать вверх. Именно таким способом дисплей использовали все программы CP/M и большинство программ MS-DOS, а операционная система UNIX до сих пор с гордостью поддерживает эту традицию.

Интересно, что набор символов ASCII не является столь уж неуместным при работе с ЭЛТ. Изначально эта кодировка включала код 1Bh (Escape), специально предназначенный для расширения набора отображаемых символов. В 1979 году Американский национальный институт стандартов опубликовал стандарт «Дополнительные управляющие символы для использования с ASCII». Его цель — «удовлетворение ожидаемых потребностей в контроле над вводом и выводом на двумерных устройствах отображения информации, включая интерактивные терминалы с электронно-лучевой трубкой и принтеры».

Код 1Bh занимает всего один байт, поэтому может иметь только одно значение. Код Escape предваряет последовательности символов, которые выполняют определенные функции. Например, последовательность, состоящая из кода Escape, за которой следуют символы [2J, стирает все содержимое экрана и перемещает курсор в верхний левый угол.

```
1Bh 5Bh 32h 4Ah
```

Телетайпный аппарат не позволяет сделать ничего подобного. Последовательность, состоящая из кода Escape, за которой следуют символы [5; 29H, перемещает курсор на пятую строку 29-го столбца.

```
1Bh 5Bh 35h 3Bh 32h 39h 48h
```

Комбинация ЭЛТ и клавиатуры, которая реагирует на ASCII-коды (возможно, и на Escape-последовательности), поступающие с удаленного компьютера, иногда называется *«немым» терминалом*. Такой терминал работает быстрее и гибче по сравнению с телетайпными аппаратами, однако его скорости недостаточно для внедрения нового в пользовательском интерфейсе. Настоящие инновации были реализованы в 1970-х годах в небольших компьютерах, в которых, как и в гипотетическом компьютере из главы 21, видеопамять была частью адресного пространства микропроцессора.

Первым признаком того, что домашние компьютеры будут сильно отличаться от своих более крупных и дорогих предшественников, было, вероятно, приложение VisiCalc, разработанное Даниэлем Бриклином (род. 1951) и Бобом Фрэнкстоном (род. 1949) в 1979 году для компьютера Apple II. Приложение VisiCalc выводило на экран двумерное изображение электронной таблицы.

До этого электронная таблица представляла собой разлинованный лист бумаги, обычно используемый для выполнения вычислений. Приложение VisiCalc заменило бумагу экраном, позволив пользователю перемещаться по электронной таблице, вводить числа и формулы, а также пересчитывать результаты после внесения изменений.

Удивительным в VisiCalc было то, что это приложение *нельзя было воссоздать на больших компьютерах*. Подобным программам необходимо очень быстро обновлять экран. По этой причине приложение VisiCalc записывало данные непосредственно в видеопамять компьютера Apple II, являющуюся частью адресного пространства микропроцессора. Интерфейс между большим компьютером с системой распределения времени и «немым» терминалом работал недостаточно быстро для такой электронной таблицы.

Чем быстрее компьютер может реагировать на сигналы клавиатуры и обновлять видеоизображение, тем плотнее его взаимодействие с пользователем. Бóльшая часть программ, написанных в течение первого десятилетия после выхода IBM PC (в 1980-х), предполагала запись данных непосредственно в видеопамять. Поскольку стандартов аппаратного обеспечения, введенных компанией IBM, придерживались и другие производители компьютеров, разработчики программ могли обойти операционную систему и напрямую задействовать аппаратное обеспечение, не опасаясь, что на некоторых компьютерах их программы будут работать неправильно (вообще не будут работать). Если бы во всех клонах IBM PC применялись разные аппаратные интерфейсы для видеодисплеев, то программистам было бы сложно учесть особенности различных конструкций.

В большинстве ранних приложений для IBM PC на экране отображался только текст без графики, что обеспечивало максимальную скорость работы. Когда видеодисплей устроен так, как описано в главе 21, программа может отобразить на экране конкретный символ, просто записав в память соответствующий ASCII-код. Программе, использующей графический видеодисплей, обычно требуется записать в память восемь или более байтов для вывода на экран изображения текстового символа.

Переход от отображения текста к отображению графики стал чрезвычайно важным шагом в эволюции. Однако процесс развития компьютерного оборудования и программного обеспечения, работающего не только с текстом, но и с графическими изображениями, происходил очень медленно. Еще в 1945 году Джон фон Нейман рассматривал возможность вывода графических изображений на дисплей, основанный на принципе работы осциллографа. Однако только в начале 1950-х годов компьютерная графика стала реальностью, когда в Массачусетском технологическом институте

(при содействии компании IBM) была учреждена Лаборатория Линкольна, перед которой стояла задача разработать компьютер для системы ПВО ВВС США. Этот проект под названием SAGE (Semi-Automatic Ground Environment, «полуавтоматическая наземная среда») подразумевал использование графических экранов, с помощью которых операторы могли бы анализировать большие объемы информации.

Первые видеодисплеи, использованные в таких системах, как SAGE, не были похожи на мониторы современных персональных компьютеров. Сегодня ПК оснащены так называемыми *растровыми* дисплеями. Как и в телевизоре, изображение на устаревших ЭЛТ-мониторах состоит из серии горизонтальных растровых линий, рисуемых лучом электронной пушки, который быстро пробегает по экрану. Такой экран можно представить в виде большого прямоугольного массива точек, называемых *пикселами*. Для хранения видеоизображения в памяти компьютера выделяется целая область, в которой один или несколько битов соответствует пикселу на экране. Значения этих битов определяют степень светимости и цвет пикселов.

Например, разрешение большинства современных компьютерных дисплеев составляет как минимум 640 пикселов по горизонтали и 480 пикселов по вертикали. Их общее количество соответствует произведению этих двух чисел: 307 200. Если для каждого пиксела выделяется только один бит памяти, то его цвет ограничивается двумя вариантами, обычно черным и белым. Например, значение 0 может указывать на черный пиксел, значение 1 — на белый. Такой видеодисплей требует 307 200 бит памяти, или 38 400 байт.

Чтобы расширить диапазон отображаемых цветов, необходимо выделить для каждого пиксела большее количество битов, а это увеличивает требования видеоадаптера к памяти. Например, для кодирования оттенков серого под каждый пиксел можно предоставить один байт памяти. При таком подходе значение 00h будет соответствовать черному цвету, значение FFh — белому, а промежуточные — оттенкам серого.

Цветное изображение ЭЛТ создается с помощью трех электронных пушек, по одной для каждого из трех основных цветов: красного, зеленого и синего. (В этом можно убедиться, рассмотрев цветной экран телевизора или компьютера через увеличительное стекло; в принтерах другой набор цветов.) Сочетание красного и зеленого дает желтый цвет, красного и синего — малиновый, зеленого и синего — голубой, а комбинация всех трех основных цветов — белый.

Простейший цветной графический адаптер требует, чтобы на один пиксел приходилось три бита, по одному биту на каждый из основных цветов. Цвет пикселов может быть закодирован следующим образом.

Биты	Цвет
000	Черный
001	Синий
010	Зеленый
011	Голубой
100	Красный
101	Малиновый
110	Желтый
111	Белый

Однако такой набор подходит только для очень простых «мультяшных» изображений. Большинство цветов, встречающихся в реальном мире, — это комбинации различных *уровней* красного, зеленого и синего. Если вы выделите два байта на пиксел, то на каждый основной цвет будет приходиться пять бит (один бит останется незадействованным). Это даст 32 уровня красного, зеленого и синего, что в общей сложности составит 32 768 разных цветов. Такая схема кодирования часто называется палитрой High color («высококачественное цветовоспроизведение»), или Thousands of colors («тысяча цветов»).

Следующий шаг — использование трех байтов на пиксел, или одного байта для каждого из основных цветов. Такая схема кодирования предусматривает 256 уровней красного, зеленого и синего, что в общей сложности дает 16 777 216 различных цветов. Часто эта палитра называется True Color («истинный/настоящий цвет»), или Millions of colors («миллионы цветов»). При разрешении дисплея 640 × 480 объем требуемой памяти составляет 921 600 байт — почти целый мегабайт.

Количество битов на пиксел иногда называется *глубиной цвета* или *цветовым разрешением*. Количество различных цветов связано с количеством битов на пиксел следующим соотношением:

$$\text{Количество цветов} = 2^{\text{количество битов на пиксел}}.$$

Плата видеоадаптера имеет определенный объем памяти, что ограничивает количество возможных комбинаций разрешения и глубины цвета. Например, плата видеоадаптера с памятью один мегабайт может обеспечить разрешение 640 × 480 при глубине цвета, равной трем байтам на пиксел. Однако если вы хотите использовать разрешение 800 × 600, для такой глубины цвета имеющейся памяти окажется недостаточно, поэтому придется довольствоваться двумя байтами на пиксел.

Несмотря на то что растровые дисплеи кажутся нам вполне естественным выбором, поначалу они редко использовались, поскольку предъявляли

огромные требования к памяти. В системе SAGE употреблялись *векторные* дисплеи, больше напоминающие осциллограф, чем телевизор. Управляемая электрическим сигналом электронная пушка рисовала на экране прямые и кривые линии, которые в течение некоторого времени сохранялись на экране, что и создавало из них примитивные изображения.

Компьютеры SAGE также поддерживали использование *световых карандашей*, которые позволяли операторам изменять кадры на дисплее. Световой карандаш — это стилус с прикрепленным к одному концу проводом. С помощью специального программного обеспечения компьютер определяет точку экрана, на который указывает световой карандаш, и изменяет изображение в соответствии с его движением.

Принцип работы светового карандаша иногда ставит в тупик даже технически подкованных людей. Суть в том, что световой карандаш не *излучает*, а *улавливает* свет. Электронная схема, управляющая электронной пушкой в ЭЛТ (вне зависимости от типа используемого дисплея), также фиксирует момент, когда свет от электронной пушки попадает на световой карандаш, определяя при этом точку экрана, на которую он указывает.

Одним из тех, кто первым предвосхитил начало эры интерактивных вычислений, был Айвен Сазерленд (род. 1938), который в 1963 году продемонстрировал революционную графическую программу Sketchpad, разработанную им для компьютеров SAGE. Эта программа могла хранить описания изображений в памяти и выводить их на экран. Кроме того, она предусматривала возможность использования светового карандаша для рисования и изменения изображений на экране, причем все это отслеживалось компьютером.

Еще одним провидцем эры интерактивных вычислений был Дуглас Энгельбарт, который прочитал статью Вэнивара Буша «Как мы можем мыслить» в 1945 году, сразу после ее публикации, а пять лет спустя начал развивать новые идеи в области компьютерных интерфейсов. В середине 1960-х годов, работая в Стэнфордском исследовательском институте, Энгельбарт полностью переосмыслил концепцию устройств ввода и придумал пятиклавишную клавиатуру для ввода команд (которая так и не прижилась), а также небольшое устройство с колесиками и кнопкой, которое он назвал *мышью*. Теперь мышь практически повсеместно применяется для перемещения указателя и выбора объектов на экране.

К счастью, многие из первых энтузиастов интерактивных графических вычислений собрались в компании Xerox в то время, когда использование растровых дисплеев стало экономически целесообразным. В 1970 году Xerox в Пало-Альто учредила исследовательский центр PARC для разработки продуктов, с которыми эта компания могла бы выйти на рынок. Вероятно, самым

известным прогнозистом в центре PARC был Алан Кэй (род. 1940), который в возрасте 14 лет узнал о библиотеке микрофильмов Вана Буша из рассказа Роберта Хайнлайна, а затем придумал портативный компьютер Dynabook.

Первый крупный проект PARC — компьютер Alto, сконструированный в 1972–1973 годах. По меркам тех лет это устройство впечатляло: массивный системный блок, обработка 16-разрядных чисел, два диска емкостью по три мегабайта, 128 килобайт оперативной памяти (расширяемой до 512 килобайт) и мышь с тремя кнопками. Поскольку во времена создания Alto однокристальных 16-разрядных микропроцессоров еще не существовало, его процессор пришлось собирать из 200 отдельных интегральных микросхем.

Видеодисплей Alto был одним из нескольких элементов, выделявших этот компьютер на фоне других. Экран напоминал лист бумаги шириной 20,3 и высотой 25,4 сантиметра. Он работал в режиме растровой графики с разрешением 606 пикселов по горизонтали и 808 пикселов по вертикали (всего 489 648 пикселов). Для каждого пиксела выделялся один бит памяти, то есть он мог быть либо черным, либо белым. Всего для видеоизображения предоставлялось 64 килобайта памяти в общем адресном пространстве процессора.

Записывая данные в эту видеопамять, программа могла выводить на экран изображения или отображать текст различных шрифтов и размеров. Перемещая по столу мышь, пользователь мог позиционировать указатель и взаимодействовать с объектами на экране. В отличие от телетайпного аппарата, реагирующего на действия пользователя с задержкой, дисплей Alto — чрезвычайно емкий двумерный массив информации — позволял пользователю взаимодействовать с компьютером более непосредственно.

В конце 1970-х годов у программ, написанных для компьютера Alto, появились интересные особенности. На экране могли одновременно отображаться несколько программ, каждая в своем окне. Графические возможности компьютера позволяли выйти за рамки текста и по-настоящему воплощать мысли пользователя. В интерфейсе появились такие графические объекты, как кнопки, меню, а также небольшие изображения, называемые *иконками*. Мышь использовалась для выбора окон и активации программных функций посредством взаимодействия с графическими объектами.

Такое программное обеспечение упростило непосредственный контакт между человеком и машиной. Работа с компьютерами не ограничивалась простым перемалыванием чисел. Как выразился Дуглас Энгельбарт в своей легендарной статье, написанной в 1963 году, такие программы были разработаны «для усиления человеческого интеллекта».

Разработки центра PARC легли в основу так называемого *графического пользовательского интерфейса* (Graphic User Interface, GUI). Однако Xerox

не стала продавать компьютер Alto (который в этом случае стоил бы более 30 тысяч долларов). Прошло более десяти лет, прежде чем идеи, воплощенные в этом компьютере, нашли применение в коммерческих продуктах.

В 1979 году PARC посетили Стив Джобс и другие сотрудники компании Apple Computer, которых впечатлило увиденное. А в январе 1983 года они представили печально известный компьютер с графическим интерфейсом Lisa, за которым год спустя последовал гораздо более успешный Macintosh.

Первый компьютер Macintosh был оснащен микропроцессором Motorola 68000, ПЗУ емкостью 64 килобайта, 128 килобайтами оперативной памяти, 3,5-дюймовым дисководом (позволяющим сохранять на дискете до 400 килобайт), клавиатурой, мышью, а также монитором с разрешением 512 × 342 (всего 175 104 пикселов) и диагональю девять дюймов. Каждому пикселу соответствовал один бит памяти, он мог быть либо черным, либо белым, поэтому объем достигал лишь 22 килобайт.

Аппаратное обеспечение первого компьютера Macintosh было элегантным, но отнюдь не революционным. От других машин, присутствовавших на рынке в 1984 году, его отличала операционная система, которая в то время именовалась *системным программным обеспечением*, а позже получила название *Mac OS*.

Такие текстовые однопользовательские операционные системы, как CP/M или MS-DOS, отличаются небольшим размером и не предусматривают обширного интерфейса прикладного программирования (API). В главе 22 я уже объяснил, что от этих текстовых операционных систем в основном требовалось предоставление приложениям доступа к файловой системе. В отличие от них такая графическая операционная система, как Mac OS, занимает намного больше памяти и предоставляет сотни функций API, каждая из которых имеет описательное название.

В то время как текстовая операционная система, вроде MS-DOS, позволяла производить лишь несколько простых функций API, например отображать на экране текст в стиле телетайпных аппаратов, графическая операционная система Mac OS должна была выводить на экран *графическое изображение*. Теоретически этого можно достичь за счет реализации одной функции API, которая разрешает приложению задавать цвет пиксела с определенными горизонтальной и вертикальной координатами. Однако такой способ оказывается неэффективным, поскольку при его применении изображение формируется очень медленно.

Разумнее сделать так, чтобы операционная система предоставляла исчерпывающую систему графического программирования, состоящую из функций API для отрисовки линий, прямоугольников, эллипсов (включая окружности) и текста. Эта система должна позволять создавать сплошные или пунктирные линии, заполнять прямоугольники и эллипсы различными узорами, а для текста

задавать разные шрифты, размеры и стили начертания. Система графического программирования также должна отвечать за визуализацию этих объектов в виде набора точек на экране.

Программы, работающие под управлением графической ОС, используют одни и те же функции API для вывода изображений на экран компьютера и для их печати на принтере. Именно поэтому документ текстового редактора отображается на экране почти так же, как и на распечатке. Этот принцип называется WYSIWYG (What You See Is What You Get — «что видишь, то и получишь»).

Привлекательность графического пользовательского интерфейса отчасти в том, что различные приложения основаны на одних и тех же принципах, благодаря чему опыт, накопленный пользователем при работе с одной программой, применим и с другими. Это означает, что операционная система также должна предусматривать функции API, позволяющие приложениям реализовывать различные компоненты пользовательского интерфейса, например кнопки и меню. Вопреки распространенному убеждению, графический интерфейс облегчает работу не только пользователям, но и программистам, поскольку допускает написание программы с современным интерфейсом. Получается, нет необходимости изобретать велосипед.

Еще до появления Macintosh несколько компаний пытались разработать графическую операционную систему для IBM PC и совместимых с ним компьютеров. В некотором смысле разработчикам компании Apple было легче, поскольку они работали над аппаратным и программным обеспечением одновременно. Системное программное обеспечение компьютера Macintosh должно было поддерживать только один тип флоппи-дисковода, один дисплей и два принтера. Однако реализация графической ОС для IBM PC предполагала поддержку множества дополнительных аппаратных средств.

Несмотря на то что компьютер IBM PC был представлен всего несколькими годами ранее (в 1981-м), многие уже привыкли к приложениям MS-DOS и не были готовы от них отказываться. И поэтому было важно, чтобы графическая ОС для IBM PC предусматривала использование приложений MS-DOS наряду с приложениями, специально написанными для новой операционной системы. (Программное обеспечение для компьютера Apple II на Macintosh не работало, поскольку в нем был другой микропроцессор.)

В 1985 году компания Digital Research, создавшая CP/M, представила систему GEM (Graphical Environment Manager, «менеджер графической среды»), компания VisiCorp, занимавшаяся распространением приложения VisiCalc, выпустила среду VisiOn, а Microsoft — версию Windows 1.0, которая быстро стала первым претендентом на победу в «войне окон». Однако только в мае 1990 года, после выпуска версии Windows 3.0, эта система обратила на себя

внимание пользователей. С тех пор ее популярность значительно возросла, и сегодня Windows используется примерно в 90%. Несмотря на сходства операционных систем Windows и Mac OS, в них совершенно разные API-интерфейсы.

Теоретически графическая ОС по сравнению с текстовой требует применения только одного дополнительного аппаратного средства — графического дисплея. При этом не нужен даже жесткий диск: у первого компьютера Macintosh его не было, а система Windows 1.0 в нем не нуждалась. Версии Windows 1.0 не требовалась мышь, хотя все были согласны с тем, что это устройство значительно упрощает процесс пользования. Неудивительно, что графические пользовательские интерфейсы распространялись по мере роста производительности микропроцессоров и увеличения емкости оперативной памяти и запоминающих устройств. С функциональным обогащением графические ОС становились более ресурсозатратными: им требовалось пара сотен мегабайт на жестком диске и более 32 мегабайт оперативной памяти.

Приложения для графических ОС почти никогда не пишутся на языке ассемблера. Поначалу программы для Macintosh в основном разрабатывались на Паскале, а приложения Windows — на C. Однако программисты из центра PARC и здесь продемонстрировали совершенно иной подход. Начиная с 1972 года они занимались созданием языка Smalltalk, воплощающего концепцию *объектно-ориентированного программирования*.

Традиционно высокоуровневые языки программирования различают код (операторы, которые обычно начинаются с ключевого слова вроде set, for или if) и данные (значения переменных). Это различие, несомненно, коренится в архитектуре фон Неймана, где все делится на машинный код и обрабатываемые им данные.

В объектно-ориентированном программировании код и данные комбинируются в *объект*. Как именно данные хранятся в объекте, можно понять только по связанному с ним коду. Объекты взаимодействуют, отправляя и получая *сообщения*, которые содержат инструкции, запрашивающие информацию.

Объектно-ориентированные языки часто бывают удобными при написании приложений для графических операционных систем, поскольку позволяют работать с объектами на экране (например, с окнами и кнопками) с учетом того, как их воспринимает пользователь. Возьмем, к примеру, кнопку. Она имеет определенный размер и положение на экране, включает некоторый текст или маленькое изображение. Все это — относящиеся к объекту данные. Код, связанный с объектом, фиксирует факт «нажатия» этой кнопки пользователем с помощью клавиатуры или мыши и отправляет сообщение об этом событии.

Наиболее популярные объектно-ориентированные языки для персональных компьютеров — это расширенные версии таких традиционных языков

типа АЛГОЛ, как C и Паскаль. Самое известное объектно-ориентированное расширение языка C — C++. (Как вы помните, в языке C два плюса соответствуют оператору инкремента.) Язык C++, разработанный в основном Бьёрном Страуструпом (род. 1950) из Bell Telephone Laboratories, сначала был реализован как транслятор, который преобразовывал программу, написанную на C++, в программу на C (хотя получавшийся в результате код на языке C был практически нечитаемым). После этого преобразования программу на C можно было скомпилировать обычным способом.

Разумеется, возможности объектно-ориентированных языков не выходят за рамки возможностей традиционных языков. Однако программирование — это деятельность, связанная с решением задач, а объектно-ориентированные языки позволяют учесть различные, часто более совершенные решения. На объектно-ориентированном языке также можно (хоть это и непросто) написать программу, компилируемую как под Mac OS, так и под Windows. Такая программа не обращается напрямую к функциям API, а вместо этого использует объекты, которые вызывают эти функции. Для компиляции программы под Mac OS или Windows нужны разные определения объектов.

Программистам, работающим на небольших компьютерах, больше не требуется запускать компилятор из командной строки. Вместо этого они используют *интегрированную среду разработки* (IDE), включающую все необходимые инструменты в одной удобной программе, которая работает подобно другим графическим приложениям. Кроме того, они широко применяют метод *визуального программирования*, в рамках которого окна программ разрабатываются в интерактивном режиме с помощью мыши для организации кнопок и других компонентов.

В главе 22 я рассказывал о текстовых файлах, которые содержат только ASCII-коды символов и получаются человекочитаемыми. Во времена текстовых операционных систем текстовые файлы были идеальным средством для обмена информацией между приложениями. Одно из важных преимуществ текстовых файлов — в них можно производить поиск, то есть программа может проверить множество файлов на наличие определенной текстовой строки. Однако когда в операционной системе появляется средство для отображения текста с использованием различных шрифтов, размеров и начертаний, возможности текстового файла внезапно оказываются неадекватными. Большинство текстовых редакторов сохраняют документы в собственном двоичном формате, а для хранения графической информации такие файлы вообще не подходят.

Например, информацию о параметрах шрифта и расположении абзацев можно закодировать вместе с текстом и получить при этом пригодный для чтения текстовый файл. Для этого нужно выбрать управляющий символ, после

которого будет следовать эта информация. В формате RTF (Rich Text Format), разработанном компанией Microsoft для обмена форматированным текстом между приложениями, для выделения информации о форматировании текста используются фигурные скобки {} и обратная косая черта \.

Эта концепция доведена до предела в формате текстового файла PostScript, разработанном соучредителем компании Adobe Systems Джоном Уорноком (род. 1940). PostScript — универсальный язык графического программирования, который используется в основном для печати текста и графики на высокотехнологичных принтерах.

Возможность отображения графики на экранах персональных компьютеров непосредственно проистекает из усовершенствования и удешевления аппаратных средств. Рост быстродействия микропроцессоров, уменьшение стоимости памяти, а также увеличение разрешения и цветопередачи дисплеев и принтеров способствовали быстрому развитию компьютерной графики.

Компьютерные графические изображения, как и видеодисплеи, бывают двух типов: векторные и растровые.

Векторные изображения создаются алгоритмически с помощью прямых и кривых линий, а также заполняемых цветом фигур. Векторная графика применяется в *системах автоматизированного проектирования* (computer-aided drawing, CAD) для создания инженерных или архитектурных чертежей. Графическое изображение такого типа может быть сохранено в формате так называемого *метафайла**. Метафайл — обычный набор команд для рисования векторных графических объектов, обычно закодированных в двоичной форме.

С помощью заполненных цветом фигур, прямых и кривых линий можно изобразить проект моста, однако если вам нужен рисунок готового моста, возможностей векторной графики будет недостаточно. Потребуется реалистичная картинка, которую невозможно создать из векторных объектов.

Для этой цели лучше использовать *растровые,* или *точечные*, изображения. При применении растровой графики изображение кодируется как прямоугольный массив битов, соответствующих пикселам устройства вывода. Как и видеодисплей, растровое изображение имеет такие параметры, как разрешение (ширина и высота в пикселах) и глубина цвета (количество битов на пиксел). Каждому пикселу соответствует одинаковое количество битов.

---

* Строго говоря, между векторной и метафайловой графикой имеется различие: описание изображения в векторной графике состоит из элементов изображения (точек, линий, полигонов), в то время как изображение в метафайловой графике — это набор команд, с помощью которых изображение сформировано. *Прим. науч. ред.*

Несмотря на то что растровое изображение двумерно, сам битовый массив — это единый поток байтов, в котором последовательно закодированы все строки пикселов.

Одни растровые изображения создаются «вручную» в специальном графическом редакторе, другие — алгоритмически с помощью компьютерного кода. Однако в наши дни растровая графика чаще всего необходима при работе со снимками реальных объектов (например, с фотографиями). Для перенесения таких изображений из реального мира в компьютер существуют специальные аппаратные средства. Как правило, они используют полупроводниковый *прибор с зарядовой связью* (ПЗС), который вырабатывает электрический ток при облучении светом. Для создания одного пиксела требуется одна ячейка ПЗС.

Старейшее из подобных устройств — *сканер*. Принцип его работы аналогичен принципу работы копировального аппарата и заключается в перемещении линейного массива элементов ПЗС вдоль поверхности сканируемого изображения, например фотографии. Элементы ПЗС генерируют электрические заряды в зависимости от интенсивности света, отраженного от поверхности изображения. Сопровождающее сканер программное обеспечение преобразует полученные сигналы в битовый массив и сохраняет его в файле.

В видеокамерах для захвата изображений используется двумерный массив ячеек ПЗС. Как правило, они записываются на видеопленку*. Однако выходной видеосигнал может подаваться непосредственно в *устройство для захвата кадра* — на плату, преобразующую аналоговый видеосигнал в массив битов. Эти устройства можно использовать с любым обычным источником видеосигнала, например с видеомагнитофоном, проигрывателем лазерных дисков или даже с телевизором.

Цифровые камеры похожи на обычные, только вместо пленки в них используется массив элементов ПЗС для захвата изображения, которое сохраняется непосредственно в памяти камеры, а затем передается в компьютер.

Графическая операционная система часто предусматривает специальный формат для хранения растровых изображений. В Mac OS используется формат Paint — от программы MacPaint, в которой он был применен впервые. (Однако предпочтителен формат Macintosh PICT, допускающий хранение и растровых, и векторных изображений.) В Windows для хранения растровых изображений используется формат BMP.

Растровые изображения могут занимать довольно много места, в связи с чем возникает необходимость в разработке способа уменьшения их объема. Этой задаче посвящена целая область информатики под названием *сжатие данных*.

---

* Так было на момент написания книги. *Прим. науч. ред.*

Вернемся к примеру с изображением, где на пиксел приходится по три бита. Это фотография неба и дома с газоном с большими областями синего и зеленого цветов. Возможно, верхняя строка растрового изображения содержит 72 синих пиксела, идущих подряд. Чтобы уменьшить объем растрового изображения, нужно закодировать в файле данные о том, что синий пиксел повторяется 72 раза. Такой способ сжатия называется *кодированием серий последовательностей* (Run-Length Encoding, RLE).

Алгоритм RLE используется в обычном офисном факсимильном аппарате для уменьшения размера изображения перед его отправкой по телефонной линии. Поскольку факс распознаёт только черный и белый цвета без оттенков серого, факсимильное изображение часто содержит длинные последовательности белых пикселов.

Уже довольно давно популярен формат растровых изображений GIF (Graphics Interchange Format, «формат для обмена изображениями»), разработанный компанией CompuServe в 1987 году. Этот формат использует алгоритм сжатия данных LZW, названный так по именам создателей (Lempel, Ziv, Welch, алгоритм Лемпеля — Зива — Велча). Алгоритм LZW является более мощным, чем RLE, поскольку помимо последовательностей одинаковых пикселов он способен распознавать *закономерности*, состоящие из различных пикселов.

Алгоритмы RLE и LZW называются методами сжатия данных *без потерь*, поскольку исходный файл может быть полностью восстановлен. Другими словами, сжатие *обратимо*. Однако метод обратимого сжатия данных подходит не для всех типов файлов. В некоторых случаях объем сжатого файла превышает объем исходного!

В последние годы широкое распространение получили методы сжатия данных *с потерями*. Алгоритм сжатия с потерями не является обратимым, поскольку при его использовании некоторые исходные данные безвозвратно теряются. Вы вряд ли примените этот алгоритм к электронным таблицам или текстовым документам, так как в них каждое число и слово имеют значение. Однако вы, вероятно, не станете возражать против его применения к изображениям при условии, что потеря данных не окажет существенного влияния на результат. Именно поэтому методы сжатия данных с потерями основаны на психологических исследованиях, задача которых выявить, что важно для человеческого визуального восприятия, а что нет. Самые распространенные методы сжатия данных с потерями, применяемые для растровых изображений, известны под общим названием JPEG (Joint Photography Experts Group, «объединенная группа экспертов по фотографии»). Формат JPEG фактически объединяет несколько алгоритмов сжатия как с потерями, так и без.

Преобразовать метафайл в растровое изображение довольно просто. Поскольку видеопамять и битовый массив концептуально идентичны, программа, способная записать метафайл в видеопамять, также может сохранить его в битовый массив.

Преобразовать растровое изображение в метафайл не так просто, а в случае сложного изображения вообще невозможно. Один из способов решения этой задачи — *оптическое распознавание символов* (Optical Character Recognition, OCR). Оно используется, когда нужно преобразовать в ASCII-коды растровое изображение некоторого текста (например, полученную по факсу или отсканированную печатную страницу). Программное обеспечение OCR анализирует точечные последовательности и определяет, каким символам они соответствуют. Из-за алгоритмической сложности этой задачи OCR-программы обычно не дают абсолютно точного результата. Еще менее точным является программное обеспечение, предназначенное для преобразования в ASCII-коды рукописного текста.

Растровые изображения и метафайлы используются для цифрового представления визуальной информации. Звуковую информацию тоже можно преобразовать в биты и байты.

Цифровой звук произвел настоящий фурор в 1983 году благодаря появлению *компакт-диска* (compact-disk, CD), который стал самым успешным продуктом в истории потребительской электроники. Компакт-диск был разработан компаниями Philips и Sony и позволял хранить на одной стороне диска диаметром 12 сантиметров цифровые аудиофайлы общей длительностью 74 минуты. Такая длительность была выбрана для того, чтобы на компакт-диске могла уместиться Девятая симфония Бетховена.

Для кодирования звука подходит метод *кодово-импульсной модуляции*. Несмотря на сложное название, концептуально этот метод довольно прост.

Звук — это вибрация. Вибрация наших голосовых связок, музыкальных инструментов, падающего в лесу дерева приводит в движение молекулы воздуха, при этом он начинает периодически (с частотой несколько сотен или тысяч раз в секунду) сжиматься и расширяться. Вибрирующий воздух, в свою очередь, воздействует на наши барабанные перепонки, и мы слышим звук.

В 1877 году Томас Эдисон изобрел фонограф, в котором для записи и воспроизведения звука на поверхности цилиндра, покрытого фольгой, создавались углубления, повторяющие форму звуковых волн. До появления компакт-диска эта техника записи звука практически не менялась, хотя цилиндры были заменены дисками, а оловянная фольга — сначала воском, а затем пластиком. Первые фонографы были полностью механическими, однако со временем для усиления звука в них стали использоваться электрические компоненты.

Переменный резистор в микрофоне преобразует звук в электрический сигнал, а электромагнит в динамике превращает этот сигнал обратно в звук.

Электрический ток, с помощью которого кодируется звук, не похож на цифровые сигналы, о которых мы говорили. Звуковое давление непрерывно изменяется, в связи с чем меняется и напряжение. Электрический ток — *аналог* звуковой волны. Для преобразования аналогового сигнала в цифровой потребуется специальный *аналого-цифровой преобразователь* (АЦП), обычно реализуемый в виде микросхемы. Выходные цифровые сигналы АЦП, количество которых равно 8, 12 или 16, обозначают относительный уровень напряжения. Например, 12-битный АЦП преобразует входящий сигнал в число от 000h до FFFh, различая при этом 4096 уровней напряжения.

При использовании метода *кодово-импульсной модуляции* напряжение, соответствующее звуковой волне, преобразуется в цифровые значения с постоянной скоростью. Эти значения сохраняются на компакт-диске в виде маленьких углублений, вырезанных на поверхности, и считываются лазерным лучом, отраженным от его поверхности. Во время воспроизведения эти значения снова конвертируются в электрический сигнал с помощью *цифро-аналогового преобразователя* (ЦАП). (ЦАП также используется в цветных графических адаптерах для преобразования значений пикселов в аналоговые сигналы, подающиеся на монитор.)

Аналоговый звуковой сигнал переводится в цифровой с постоянной скоростью, называемой *частотой дискретизации*. В 1928 году Гарри Найквист из Bell Telephone Laboratories показал, что частота дискретизации должна как минимум в два раза превышать максимальную частоту звука, который необходимо записать и воспроизвести. Считается, что человек воспринимает звуки в диапазоне частот от 20 до 20 000 герц. Частота дискретизации, используемая при записи компакт-дисков, более чем вдвое превышает максимальный показатель и составляет 44 100 герц.

Разрядность (количество бит на выборку) определяет динамический диапазон компакт-диска — разницу между самым громким и самым тихим звуком, который можно записать и воспроизвести. Это требует пояснения: будучи аналогом звуковой волны, электрический сигнал отклоняется от нулевого значения; максимальное отклонение — *амплитуда* волны. *Интенсивность* звука пропорциональна удвоенной амплитуде. Десятикратному увеличению интенсивности звука соответствует один *бел* (единица измерения относительной силы звука, названная в честь Александра Белла); один *децибел* — 0,1 бела, примерно минимальное увеличение интенсивности звука, которое человек в состоянии воспринять.

Динамический диапазон для 16-разрядного звука — 96 децибел, что примерно соответствует разнице между порогом слышимости (за которым мы

ничего не слышим) и болевым. Именно такая разрядность используется при записи компакт-дисков.

Таким образом, каждой секунде звуковой записи на компакт-диске соответствует 44 100 выборок по два байта. Если вы предпочитаете стереозвук, необходимо удвоить это число, чтобы в итоге получить 176 400 байт на секунду. Это 10 584 000 байт на минуту звукозаписи. (Теперь вы понимаете, почему цифровая звукозапись распространилась только в 1980-е годы.) Для записи на компакт-диске стереозвука длительностью 74 минуты требуется 783 216 000 байтов.

Цифровой звук по сравнению с аналоговым обладает многими хорошо известными преимуществами. В частности, при копировании аналогового звука (например, при записи на фонограф звука, воспроизводимого с магнитной ленты) его качество ухудшается. Оцифрованный звук закодирован числами, которые всегда можно скопировать без потери точности. В аналоговых телефонных сетях качество звука было тем хуже, чем длиннее расстояние, преодолеваемое телефонным сигналом. Сейчас этой проблемы не существует. Поскольку бóльшая часть телефонной системы перешла на цифровой звук, звонки с другого конца страны по качеству не уступают звонкам с соседней улицы.

На компакт-дисках можно хранить не только звук, но и другие данные. Диск, используемый исключительно для хранения данных, называется CD-ROM (CD Read-Only Memory, «память только для чтения»). Как правило, емкость таких дисков — около 660 мегабайт. В настоящее время большинство компьютеров оснащено специальными дисководами, а диски — распространенные носители для коммерческого программного обеспечения и игр.

Средства для работы со звуком и видео, добавленные в персональный компьютер, получили название «*мультимедиа*». Теперь они настолько популярны, что не нуждаются в специальном названии. Большинство современных домашних компьютеров оснащены звуковой платой, которая включает устройство АЦП для записи звука через микрофон и устройство ЦАП для воспроизведения записанного звука через динамики. Звуки могут храниться на диске в формате WAV (waveform — «в форме волны»).

Поскольку при записи и воспроизведении звука на домашних компьютерах CD-качество требуется не всегда, программы для Macintosh и Windows предусматривают более низкие значения частоты дискретизации (22 050, 11 025 и 8000 герц) разрядности (восемь бит), а также возможность создания монофонической записи. Таким образом, на одну секунду звучания может приходиться 8000 байт, или 480 тысяч байт на одну минуту.

Все, кто смотрел научно-фантастические фильмы, знают, что компьютеры будущего общаются с пользователями на человеческом языке. Если компьютер оснащен аппаратными средствами для записи и воспроизведения звука,

то решение всех остальных задач сводится к написанию программного обеспечения.

Существует несколько способов научить компьютер употреблять узнаваемые слова и предложения при общении с пользователем. Один из них состоит в записи произнесенных человеком фрагментов предложений, фраз, слов и чисел, которые затем можно сохранить в файлах и комбинировать. Этот подход часто применяется в информационных системах, доступ к которым осуществляется по телефону, и он отлично работает при ограниченном количестве комбинаций воспроизводимых слов и чисел.

Более общий способ синтеза человеческой речи предполагает преобразование произвольного текста в кодировке ASCII в звуковой файл. Поскольку написание слов иногда отличается от их произношения, программа может использовать словарь или сложные алгоритмы для определения правильного произношения. Из простых звуков (называемых фонемами) можно составлять целые слова. Часто программе требуются и другие корректировки. Например, если в конце предложения стоит знак вопроса, то последнее слово нужно произнести более высоким голосом.

Распознавание голоса, или преобразование звука в ASCII-коды, — более сложная задача. Многим трудно воспринимать даже диалекты родного языка. Несмотря на то что программы для распознавания речи существуют, им необходима некоторая тренировка, прежде чем они смогут качественно расшифровывать речь конкретного пользователя. Преобразование речи в ASCII-коды — довольно простая задача по сравнению с тем, чтобы научить компьютер по-настоящему «понимать» сказанное. Эта проблема относится к области *искусственного интеллекта*.

Звуковые карты современных компьютеров также снабжены небольшими электронными синтезаторами, которые могут имитировать звучание 128 мелодических и 47 ударных инструментов. Они называются MIDI-синтезаторами* (Musical Instrument Digital Interface — «цифровой интерфейс для музыкальных инструментов»). Спецификация MIDI была разработана в начале 1980-х годов консорциумом производителей электронных музыкальных синтезаторов для подключения этих электронных устройств к компьютерам и друг к другу.

В различных типах MIDI-синтезаторов используются разные способы синтеза звука музыкальных инструментов, некоторые реалистичнее, чем другие. Общее качество звука, создаваемого конкретным MIDI-синтезатором, не имеет отношения к спецификации MIDI. Все, что требуется от синтезатора, — воспроизведение

* На сегодняшний день MIDI-синтезированная музыка практически не используется. *Прим. науч. ред.*

звуков в ответ на короткие сообщения длиной один, два или три байта. Как правило, эти сообщения указывают, какой инструмент необходим, какую ноту нужно сыграть, звучание какой из нот следует прекратить.

MIDI-файл — это набор MIDI-сообщений с информацией о том, когда следует выполнять то или иное действие. Как правило, MIDI-файл содержит всю музыкальную композицию, которую воспроизводит MIDI-синтезатор. MIDI-файл обычно компактнее, чем файл в формате WAV, содержащий ту же музыку. Если говорить об относительном размере, то файл в формате WAV можно сопоставить с растровым изображением, а MIDI-файл — с векторным. Недостатком MIDI-технологии является то, что закодированная таким образом музыка может отлично звучать на одном MIDI-синтезаторе и ужасно — на другом.

Еще одно направление мультимедиа — цифровое видео. Иллюзия движения видео- и телевизионных изображений достигается путем быстрой смены отдельных неподвижных изображений, которые называются *кадрами*. Фильмы воспроизводятся со скоростью 24 кадра в секунду. Для телевидения США стандарт — скорость 30 кадров в секунду, а для большинства других — 25 кадров в секунду.

Воспроизводимый на компьютере видеофайл является просто последовательностью растровых изображений, сопровождаемых звуком. Без применения алгоритма сжатия данных размер такого файла будет огромным. Например, при разрешении 640 × 480 пикселов и 24-битной глубине цвета каждый кадр фильма занимает 921 600 байт. При скорости воспроизведения 30 кадров в секунду нам требуется 27 648 000 байт для записи одной секунды видео. При таких параметрах одна минута будет занимать 1 658 880 000 байт, а весь двухчасовой фильм — 199 065 600 000 байт — около 200 гигабайт. Так что большинство воспроизводимых на персональном компьютере фильмов имеют небольшое разрешение и невысокое качество, и они короткие*.

Алгоритм сжатия данных JPEG уменьшает размер неподвижных изображений, а алгоритм MPEG (Motion Pictures Expert Group — «экспертная группа по движущимся изображениям») — фильмов. Технология сжатия движущихся изображений основана на факте, что смежные кадры обычно содержат много одинаковой информации.

Существуют несколько стандартов MPEG. MPEG-2 предназначен для телевидения высокой четкости (HDTV) и *цифровых видеодисков* (DVD, Digital Video Disks) или *цифровых многоцелевых дисков* (Digital Versatile Disc). DVD-диск имеет такой же размер, что и компакт-диск, однако данные можно записывать

* С ростом пропускной способности сетей связи и быстродействия компьютеров, с появлением стандартов HD и BlueRay ситуация стала совершенно иной. *Прим. науч. ред.*

на обеих его сторонах по два слоя. Видео, записанное на DVD-диске, сжимается примерно в 50 раз, поэтому для двухчасового фильма требуется всего четыре гигабайта, и оно может уместиться на одном слое одной стороны. Использование обоих слоев и обеих сторон увеличивает емкость DVD-дисков примерно до 16 гигабайт, что примерно в 25 раз превышает емкость компакт-диска.

Являются ли диски CD-ROM и DVD-ROM современной реализацией устройства Memex, описанного Вэниваром Бушем? Изначально в Memex планировалось использовать микрофильмы, однако диски CD-ROM и DVD-ROM гораздо больше подходят на роль носителей информации для этого устройства. По сравнению с физическими преимущество электронных носителей следующее: в них легче найти нужные данные. К сожалению, мало кто может получить доступ сразу к нескольким CD- или DVD-дискам. Максимально приближенная к концепции Буша реализация предполагает не хранение всей нужной информации в одном месте, а *соединение* компьютеров друг с другом для обмена информацией и более эффективного использования запоминающих устройств.

Первым человеком, которому удалось осуществить удаленное управление компьютером, был Джордж Стибиц, сотрудник Bell Labs, который в 1930-х годах разработал релейный компьютер. Удаленное управление этим компьютером имело место в 1940 году на демонстрации в Дартмуте.

Телефонная система устроена так, чтобы передавать по проводам звук, а не биты. Передача битов по телефонным проводам требует их преобразования в звук. Непрерывная звуковая волна постоянной частоты и амплитуды, называемая *несущей*, вообще не передает существенной информации. Стоит изменить какой-то из параметров этой звуковой волны, другими словами, произвести *модуляцию*, чтобы один из ее параметров колебался между двумя разными состояниями, и вы сможете представить с ее помощью значения 0 и 1. Преобразование битов в звук происходит с помощью устройства, называемого *модемом* (modem; модулятор/демодулятор). Модем — последовательный интерфейс, поскольку отдельные биты байта передаются друг за другом, а не одновременно. (Принтеры часто подключаются к компьютерам через параллельный интерфейс, позволяющий одновременно передавать целый байт благодаря наличию восьми проводов.)

В ранних модемах использовался метод *частотной манипуляции*. Модем, передающий данные со скоростью, например, 300 бит в секунду, может преобразовывать 0 бит в частоту 1070 герц, а один бит — в частоту 1270 герц. Каждый байт предваряется старт-битом и заканчивается стоп-битом, поэтому для передачи одного байта требуется десять бит. Первые модемы передавали данные со скоростью 300 бит (30 байт) в секунду. В современных модемах

используются более сложные технологии, позволяющие увеличить скорость передачи данных более чем в 100 раз.

На заре эры компьютерных коммуникаций энтузиасты объединяли персональный компьютер и модем в *электронные доски объявлений*, к которым через телефонную линию могли подключаться пользователи других компьютеров и скачивать файлы, то есть копировать файлы с удаленного компьютера на свой. Эта концепция широко применялась и такими крупными информационными сервисами, как CompuServe. В большинстве случаев обмен данными осуществлялся в форме ASCII-кодов.

Качественное отличие интернета от этих ранних систем заключается в децентрализованности. Работа интернета основана на наборе протоколов, с помощью которых компьютеры взаимодействуют друг с другом. Самый важный — семейство протоколов TCP/IP (Transmission Control Protocol / Internet Protocol). Вместо передачи ASCII-кодов по проводам передатчики, работающие на основе TCP/IP, разделяют крупные блоки данных на небольшие *пакеты*, которые отправляются по линии передачи (например, по телефонной) и заново собираются в приемнике.

Популярная составляющая интернета — сервис World Wide Web (WWW, Всемирная паутина), использующий протокол HTTP (Hypertext Transfer Protocol — «протокол передачи гипертекста»). Содержимое веб-страниц оформляется в текстовом формате HTML (Hypertext Markup Language — «язык разметки гипертекста»). Слово «гипертекст» используется для описания совокупности связанных фрагментов (как в устройстве Memex Вэнивара Буша). HTML-файл может содержать ссылки на другие веб-страницы, на которые с него можно легко перейти.

Формат HTML напоминает описанный ранее RTF, поскольку содержит ASCII-текст вместе с информацией о его форматировании. HTML также позволяет ссылаться на изображения в форматах GIF, PNG (Portable Network Graphics) и JFIF (JPEG File Interchange Format). Большинство веб-браузеров разрешают просматривать HTML-файлы именно благодаря их текстовому формату. Еще одно преимущество текстового представления HTML-файла — легкость, с которой в нем можно осуществлять поиск. Несмотря на свое название, HTML не относится к языкам *программирования*, о которых мы говорили в главах 19 и 24. Веб-браузер считывает данные из HTML-файла и соответствующим образом форматирует текст и графику.

Иногда при просмотре определенных веб-страниц необходимо запустить специальный программный код. Такой код может работать либо на *сервере,* где хранятся исходные веб-страницы, либо на *клиенте,* то есть на вашем компьютере. На сервере вся необходимая работа (например, интерпретация содержимого полей онлайн-формы), как правило, выполняется с помощью сценариев

CGI (Common Gateway Interface — «общий интерфейс шлюза»). Запускаемый на стороне клиента код обычно содержится в HTML-файле в виде сценария, написанного на простом языке программирования JavaScript. Веб-браузер интерпретирует операторы JavaScript так же, как текст HTML.

Почему веб-сайт не может просто предоставить исполняемую программу для запуска на компьютере? Во-первых, многое зависит от типа машины. Компьютеру Macintosh требуется исполняемый файл, содержащий машинный код для процессора PowerPC и обращения к функциям API Mac OS, PC-совместимому — исполняемый файл, содержащий код для процессора Intel Pentium и обращения к функциям API ОС Windows. Однако существуют другие компьютеры и графические операционные системы. Более того, вам вряд ли захочется загружать все исполняемые файлы без разбора, поскольку можно загрузить из ненадежного источника файл, который способен причинить вред.

Для решения этой проблемы компания Sun Microsystems разработала язык Java (не путайте с JavaScript). Java — это полноценный объектно-ориентированный язык программирования, похожий на C++. В предыдущей главе я объяснил, в чем разница между компилируемыми и интерпретируемыми языками. Язык Java — что-то среднее. Программу, написанную на Java, необходимо скомпилировать, но результатом компиляции обычно является не машинный код, а *байт-коды Java*. По структуре они похожи на машинный код, но предназначены для воображаемого компьютера, называемого *виртуальной машиной Java* (JVM, Java virtual machine). Компьютер, на котором выполняется скомпилированная Java-программа, эмулирует работу JVM, интерпретируя байт-коды. Java-программа использует установленную графическую операционную систему, что позволяет заниматься *платформонезависимым* программированием.

Несмотря на то что бóльшая часть этой книги была посвящена использованию электричества для передачи сигналов и информации по проводам, более эффективна передача данных в виде световых импульсов по оптоволоконному кабелю — тонкой стеклянной или полимерной трубке, позволяющей свету огибать углы. При использовании такой технологии скорость передачи данных достигает миллиардов бит в секунду.

Таким образом, в будущем именно фотоны, а не электроны будут доставлять бóльшую часть информации в наши дома и офисы. Это будет напоминать многократно ускоренную передачу кода Морзе и обмен вспышками света, которые мы когда-то использовали для того, чтобы поделиться полуночной мудростью с лучшим другом, жившим в доме напротив.

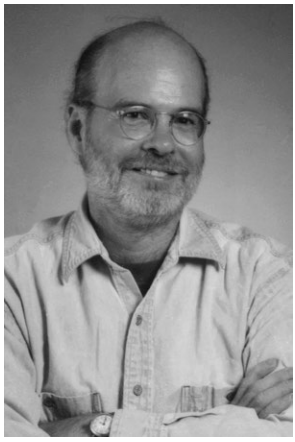
Благодарности

Идея написать эту книгу возникла у меня в 1987 году. Я обдумывал ее на протяжении десяти лет и писал наброски в файле Microsoft Word с января 1996 года по июль 1999-го. Я выражаю огромную благодарность:

— читателям первых черновиков книги Шерил Кантер, Йену Истлунду, Питеру Голдеману, Линн Магалска и Дейрдре Синнотт за комментарии, замечания и предложения;
— моему агенту Клодетт Мур из литературного агентства Moore Literary Agency и всем сотрудникам Microsoft Press, которые помогли этой книге увидеть свет;
— моей матери, которая всегда поддерживала меня в начинаниях;
— маленькой кошечке, жившей у меня с 1982 по май 1999 года, благодаря которой в книге появились многие примеры;
— таким веб-сайтам, как Bibliofind (*bibliofind.com*) и Advanced Book Exchange (*abebooks.com*), предоставляющим удобный доступ к подержанным книгам, а также сотрудникам отдела науки, техники и бизнеса Нью-Йоркской публичной библиотеки (*nypl.org*);
— моим друзьям, без поддержки которых мой замысел не был бы реализован;
— и еще раз Дейрдре, моему идеальному читателю и не только.

Чарльз Петцольд
15 июля 1999 года

Об авторе



Чарльз Петцольд живет в Нью-Йорке и занимается программированием и написанием книг о персональных компьютерах. Его классическая книга «Программирование для Windows» выдержала множество изданий и оказала значительное влияние на целое поколение программистов. Книга «Код» позволяет читателю с любым уровнем технической подготовки узнать, как работают компьютеры, и познакомиться с одаренным преподавателем.

Библиография

Аннотированную библиографию для этой книги можно найти на сайте *charlespetzold.com/code*.

Где купить наши книги

Специальное предложение для компаний

Если вы хотите купить сразу более 20 книг, например для своих сотрудников или в подарок партнерам, мы готовы обсудить с вами специальные условия работы. Для этого обращайтесь к нашему менеджеру по корпоративным продажам: +7 (495) 792-43-72, b2b@mann-ivanov-ferber.ru

Книготорговым организациям

Если вы оптовый покупатель, обратитесь, пожалуйста, к нашему партнеру — торговому дому «Эксмо», который осуществляет поставки во все книготорговые организации.
142701, Московская обл., г. Видное, Белокаменное ш., д. 1;
+7 (495) 411-50-74; reception@eksmo-sale.ru

Адрес издательства «Эксмо» 125252, Москва, ул. Зорге, д. 1; +7 (495) 411-68-86; info@eksmo.ru / www.eksmo.ru

Санкт-Петербург
СЗКО Санкт-Петербург, 192029, Санкт-Петербург, пр-т Обуховской Обороны, д. 84е;
+7 (812) 365-46-03 / 04; server@szko.ru

Нижний Новгород
Филиал «Эксмо» в Нижнем Новгороде, 603094, Нижний Новгород, ул. Карпинского, д. 29;
+7 (831) 216-15-91, 216-15-92, 216-15-93, 216-15-94;
reception@eksmonn.ru

Ростов-на-Дону
Филиал «Эксмо» в Ростове-на-Дону, 344023, Ростов-на-Дону, ул. Страны Советов, 44а;
+7 (863) 303-62-10;
info@rnd.eksmo.ru

Самара
Филиал «Эксмо» в Самаре, 443052, Самара, пр-т Кирова, д. 75/1, лит. «Е»;
+7 (846) 269-66-70 (71…73);
RDC-samara@mail.ru

Екатеринбург
Филиал «Эксмо» в Екатеринбурге, 620024, Екатеринбург, ул. Новинская, д. 2щ;
+7 (343) 272-72-01 (02...08)

Новосибирск
Филиал «Эксмо» в Новосибирске, 630015, Новосибирск, Комбинатский пер., д. 3;
+7 (383) 289-91-42;
eksmo-nsk@yandex.ru

Хабаровск
Филиал «Эксмо-Новосибирск» в Хабаровске, 680000, Хабаровск, пер. Дзержинского, д. 24, лит. «Б», оф. 1; +7 (4212) 910-120; eksmo-khv@mail.ru

Казахстан
«РДЦ Алматы», 050039, Алматы, ул. Домбровского, д. 3а; +7 (727) 251-59-89 (90, 91, 92);
RDC-almaty@eksmo.kz

Украина
«Эксмо-Украина», Киев, ООО «Форс Украина», 04073, Киев, Московский пр-т, д. 9; +38 (044) 290-99-44;
sales@forsukraine.com

Если у вас есть замечания и комментарии к содержанию, переводу, редактуре и корректуре, то просим написать на be_better@m-i-f.ru, так мы быстрее сможем исправить недочеты.

НЕЙРОБИОЛОГИЯ

ТЕОРИЯ ИГР

ЛИНГВИСТИКА

ЭКОНОМИКА

АСТРОФИЗИКА

И МНОГОЕ ДРУГОЕ

МИФ Научпоп

Весь научпоп на одной странице:
mif.to/science

Узнавай первым о новых книгах, скидках и подарках из нашей рассылки
mif.to/sci-letter

#mifnauka

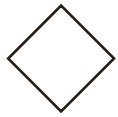
Научно-популярное издание

Чарльз **Петцольд**
Код
Тайный язык информатики

Руководитель редакции *Артем Степанов*
Шеф-редактор направления
«Переводная литература» *Ренат Шагабутдинов*
Ответственный редактор *Наталья Довнар*
Редактор *Оксана Салова*
Арт-директор *Алексей Богомолов*
Обложка *Наталия Майкова*
Верстка *Елена Бреге*
Корректоры *Олег Пономарев, Дарья Балтрушайтис, Анна Угрюмова*

ООО «Манн, Иванов и Фербер»
www.mann-ivanov-ferber.ru
www.facebook.com/mifbooks
www.vk.com/mifbooks

16+



Культовая книга талантливого преподавателя стала для многих первым уверенным шагом в программировании.

От кодов до азбуки Морзе и шрифта Брайля, от устройства фонариков до телеграфов и реле… Автор заглянул в XIX век и на примере простейших электрических компонентов объяснил устройство компьютера.

***

«Хотя в настоящее время компьютеры сложнее, чем четверть или полвека назад, они не изменились фундаментально. Вот почему изучать историю техники так здорово: чем сильнее углубляешься в прошлое, тем проще становятся технологии. Поэтому легко добраться до точки, где понятно решительно все… Хочу, чтобы с помощью „Кода“ вы научились разбираться во всех этих вещах настолько, чтобы смогли потягаться с электротехниками и программистами».

Чарльз Петцольд

ISBN 978-5-00117-545-2

издательство
МАНН, ИВАНОВ И ФЕРБЕР

Максимально полезные книги
на сайте **mann-ivanov-ferber.ru**

Like facebook.com/**mifbooks**
vk.com/**mifbooks**
instagram.com/**mifbooks**