

НОАМ НИСАН • ШИМОН ШОКЕН

Архитектура компьютерных систем

КАК СОБРАТЬ СОВРЕМЕННЫЙ КОМПЬЮТЕР
ПО ВСЕМ ПРАВИЛАМ



КЛАССИКА **ИТ**

NOAM NISAN • SHIMON SCHOCKEN

The Elements of Computing Systems

BUILDING A MODERN COMPUTER
FROM FIRST PRINCIPLES

НОАМ НИСАН • ШИМОН ШОКЕН

Архитектура КОМПЬЮТЕРНЫХ СИСТЕМ

КАК СОБРАТЬ СОВРЕМЕННЫЙ КОМПЬЮТЕР
ПО ВСЕМ ПРАВИЛАМ

УДК 004.2
ББК 32.973.26
Н69

The Elements of Computing Systems: Building a Modern Computer from First Principles

Noam Nisan, Shimon Schocken

© 2021 Massachusetts Institute of Technology

© 2023 Eksmo Publishing House

The rights to the Russian-language edition obtained through
Alexander Korzhenevski Agency (Moscow).

Нисан, Ноам.

Н69 Архитектура компьютерных систем. Как собрать современный компьютер по всем правилам / Ноам Нисан, Шимон Шокен ; [перевод с английского О. И. Перфильева]. — Москва : Эксмо, 2023. — 496 с. — (Классика ИТ. Главные книги для программистов).

ISBN 978-5-04-181053-5

Лучший способ понять, как работают компьютеры, — это построить один из них с нуля! Так считают авторы этой книги и потому предлагают практический подход к изучению компьютерных систем. Внутри вас ждет не только исчерпывающее теоретическое описание работы современного компьютера, но и алгоритм конкретных шагов, необходимых для его конструирования. В отличие от других учебников, которые охватывают только один аспект темы, эта книга дает целостное и исчерпывающее знание прикладной информатики, необходимое для создания собственных проектов.

УДК 004.2
ББК 32.973.26

ISBN 978-5-04-181053-5

© Перфильев О.И., перевод на русский язык, 2023
© Оформление. ООО «Издательство «Эксмо», 2023

*Посвящается нашим родителям,
научившим нас, что меньше — это больше.*

Оглавление

ПРЕДИСЛОВИЕ	11
Предмет (темы) книги	14
Курсы	15
Ресурсы	17
Структура	18
Проекты	19
Второе издание	20
Благодарности	21
 I. АППАРАТНОЕ ОБЕСПЕЧЕНИЕ	 23
Привет, нижний мир	23
От Nand до «Тетриса»	25
Абстракция и реализация	29
Методология	31
Путь, который вас ожидает	33
 1. Булева логика	 35
1.1. Булева алгебра	36
1.2. Логические вентили	40
1.3. Аппаратное конструирование	44
1.4. Спецификация	50
1.5. Реализация	58
1.6. Проект	62
1.7. Перспектива	64
 2. Булева арифметика	 67
2.1. Арифметические операции	68
2.2. Двоичные числа	68
2.3. Двоичное сложение	71
2.4. Двоичные числа со знаком	72
2.5. Спецификация	74
2.6. Реализация	82
2.7. Проект	84
2.8. Перспектива	85

3. Память	87
3.1. Устройства памяти	88
3.2. Секвенциальная логика	90
3.3. Спецификация	97
3.4. Реализация	102
3.5. Проект	107
3.6. Перспектива	108
4. Машинный язык	111
4.1. Машинный язык: обзор	113
4.2. Машинный язык Nask	118
4.3. Программирование на языке Nask	135
4.4. Проект	138
4.5. Перспектива	141
5. Компьютерная архитектура	143
5.1. Основы компьютерной архитектуры	144
5.2. Аппаратная платформа Nask: спецификация	152
5.3. Реализация	160
5.4. Проект	165
5.5. Перспектива	168
6. Ассемблер	173
6.1. Общие принципы	174
6.2. Спецификация машинного языка Nask	177
6.3. Перевод с языка ассемблера в двоичный код	180
6.4. Реализация	182
6.5. Проект	188
6.6. Перспектива	191
II. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ	193
II.1. Примеры программирования на языке Jack	195
II.2. Компиляция программы	201
7. Виртуальная машина I: обработка	205
7.1. Парадигма виртуальной машины	207
7.2. Стековая машина	210
7.3. Спецификация ВМ, часть I	217
7.4. Реализация	218
7.5. Проект	230
7.6. Перспектива	235
8. Виртуальная машина II: управление	239
8.1. Высокоуровневая магия	240
8.2. Ветвление	243
8.3. Функции	246
8.4. Спецификация ВМ, часть II	255
8.5. Реализация	257
8.6. Проект	266
8.7. Перспектива	271

9. Высокоуровневый язык	275
9.1. Примеры	277
9.2. Спецификация языка Jack	283
9.3. Написание приложений на языке Jack	296
9.4. Проект	299
9.5. Перспектива	301
10. Компилятор I: синтаксический анализ	303
10.1. Основы	305
10.2. Спецификация	316
10.3. Реализация	320
10.4. Проект	325
10.5. Перспектива	331
11. Компилятор II: генерация кода	333
11.1. Генерация кода	335
11.2. Спецификация	361
11.3. Реализация	362
11.4. Проект	374
11.5. Перспектива	379
12. Операционная система	381
12.1. Основы	383
Эффективность прежде всего	385
Умножение	386
Деление	388
Квадратный корень	390
12.2. Спецификация ОС Jack	403
12.3. Реализация	404
12.4. Проект	413
План тестирования	414
Полный тест	418
12.5. Перспектива	419
13. Веселье продолжается	421
Аппаратные реализации	422
Улучшения аппаратной части	423
Высокоуровневые языки	423
Оптимизация	424
Обмен данными	424
ПРИЛОЖЕНИЕ 1. ПОСТРОЕНИЕ БУЛЕВЫХ ФУНКЦИЙ	425
П1.1. Булева алгебра	425
П1.2. Построение булевых функций	427
П1.3. Выразительная сила Nand	429
ПРИЛОЖЕНИЕ 2. ЯЗЫК ОПИСАНИЯ АППАРАТУРЫ	433
П2.1. Основы HDL	433
П2.2. Многобитные шины	438
П2.3. Встроенные микросхемы	440

П2.4. Последовательностные микросхемы	442
П2.5. Визуализация микросхем	446
П2.6. Практический справочник по HDL	449
ПРИЛОЖЕНИЕ 3. ЯЗЫК ОПИСАНИЯ ТЕСТОВ	457
П3.1. Общие рекомендации	458
П3.2. Тестирование микросхем в симуляторе аппаратуры	461
П3.3. Тестирование программ на машинном языке в симуляторе ЦПУ	471
П3.4. Тестирование программ ВМ в эмуляторе ВМ	473
ПРИЛОЖЕНИЕ 4. НАБОР МИКРОСХЕМ НАСК	477
ПРИЛОЖЕНИЕ 5. НАБОР СИМВОЛОВ НАСК	479
ПРИЛОЖЕНИЕ 6. API ОС JACK	481
Math	481
String	482
Array	483
Output	483
Screen	484
Keyboard	485
Memory	486
Sys	486
АЛФАВИТНЫЙ УКАЗАТЕЛЬ	487

Предисловие

Услышанное я забываю; увиденное запоминаю; сделанное понимаю.

— Конфуций (551–479 до н. э.)

Обычно утверждается, что просвещенные люди XXI века должны иметь какое-то представление о ключевых концепциях, лежащих в основе так называемой четверки BANG: «биты», «атомы», «нейроны» и «гены» (*Bits, Atoms, Neurons, and Genes*). И хотя наука добилась значительных успехов в исследовании основ во всех этих сферах, вряд ли мы когда-нибудь поймем до конца, как на самом деле устроены атомы, нейроны и гены.

Чего, впрочем, нельзя сказать о «битах» и о вычислительных системах в целом: несмотря на фантастическую сложность современных компьютеров, понять, как они работают и как устроены, можно полностью. Поэтому, когда мы с благоговением смотрим на окружающий нас физический мир, приятно осознавать, что хотя бы одна сфера из четверки BANG полностью доступна для человеческого понимания.

И действительно, в первые годы существования компьютеров общее представление о работе вычислительных машин мог получить любой любопытный человек. Взаимодействие между аппаратным и программным обеспечением было достаточно простым и прозрачным, позволяющим окинуть взглядом всю картину работы компьютера. Увы, по мере усложнения цифровых технологий эта ясность была практически утрачена: самые фундаментальные идеи и технические

приемы информатики — сама суть этой сферы — теперь скрыты под множеством слоев не до конца понятных интерфейсов и проприетарных реализаций. Неизбежным следствием такого усложнения стала специализация: изучение прикладной информатики превратилось в погоню за множеством нишевых курсов, каждый из которых охватывает лишь отдельный аспект этой области.

Мы написали эту книгу, поскольку у нас сложилось впечатление, что многие студенты, изучающие информатику, не видят леса за деревьями. Типичный студент пробегает через серию курсов по программированию, теории и инженерии, не останавливаясь, чтобы оценить красоту общей картины. А картина в целом такова, что аппаратные, программные и прикладные системы тесно связаны между собой, что не очевидно через скрытую паутину абстракций, интерфейсов и контрактных реализаций.

Невозможность рассмотреть общую картину порождает у многих учащихся и профессионалов тревожное ощущение, будто они не до конца понимают, что происходит внутри компьютеров. И это прискорбно, поскольку компьютеры — самые важные машины XXI века.

Согласно нашему мнению, лучший способ понять, как работают компьютеры, — это собрать один из них с нуля. Исходя из этого, мы пришли к следующей идее: опишем простую, но достаточно мощную компьютерную систему, предложив учащимся создать ее аппаратную платформу и разработать иерархическую структуру программной системы с нуля. В процессе работы будем делать это правильно. Мы утверждаем, что создание компьютера общего назначения (или универсального компьютера) на основе базовых принципов — это очень серьезная и важная задача.

Как следствие, мы решили воспользоваться уникальной образовательной возможностью не только создать некий механизм, но и проиллюстрировать на практике основные принципы планирования и управления крупными проектами по разработке аппаратного и программного обеспечения. Кроме того, мы стремились показать захватывающий процесс создания фантастически сложных и полезных систем на основе фундаментальных принципов посредством тщательных рассуждений и модульного планирования.

Результатом этих усилий стал практический курс, известный теперь под общим названием «От Nand до “Тетриса”»: практическое путешествие, начинающееся с самого элементарного логического элемента Nand, двадцать проектов спустя заканчивающееся созданием универсальной компьютерной системы, способной запустить «Тетрис», а также любую программу, которая только придет в вашу голову. Несколько раз спроектировав, построив, перепроектировав и восстановив компьютерную систему, мы написали эту книгу в таком формате, чтобы это мог сделать любой учащийся. Мы также открыли веб-сайт, где выложили все материалы и программные инструменты нашего проекта, доступные для всех, кто хочет изучать или преподавать курс «От Nand до “Тетриса”».

Реакция была ошеломляющей. Сегодня курсы «От Nand до “Тетриса”» преподаются во многих университетах, средних учебных заведениях, учебных лагерях программирования, на онлайн-платформах и в хакерских клубах по всему миру. Наша книга и наши онлайн-курсы приобрели огромную популярность, и тысячи учащихся — от старшеклассников до инженеров Google — регулярно публикуют отзывы, в которых описывают курс «От Nand до “Тетриса”» как лучший образовательный опыт в своей жизни. Как однажды сказал Ричард Фейнман: «То, что я не могу создать, я не понимаю». Практическая программа «От Nand до “Тетриса”» как раз посвящена пониманию через создание. Очевидно, что в людях очень силен менталитет создателя и они страстно привязываются к этой идее.

После публикации первого издания книги мы получили множество вопросов, комментариев и предложений. По мере того как мы решали эти вопросы и вносили изменения в онлайн-материалы, между нашей веб-версией курса «От Nand до “Тетриса”» и книгой появились расхождения. Кроме того, мы подумали, что многие разделы книги могли бы выиграть от большей ясности изложения и от лучшей организации. Поэтому, после того как мы и так уже откладывали насколько могли данный процесс, мы решили засучить рукава и написать новый вариант данной книги, в результате чего в свет вышло это издание. В оставшейся части предисловия описывается именно оно, а в конце сравнивается с предыдущим.

Предмет (темы) книги

Книга предлагает учащимся получить значительный объем знаний в ходе выполнения ряда заданий по конструированию аппаратного и программного обеспечения. В частности, в процессе выполнения практических проектов затрагиваются следующие темы.

- *Аппаратное обеспечение*: булева арифметика, комбинационная логика, последовательностная (секвенциальная) логика, проектирование и реализация логических элементов (вентилей), мультиплексоры, триггеры, регистры, блоки оперативной памяти (ОЗУ), счетчики, язык описания аппаратуры (HDL), моделирование микросхем, верификация и тестирование микросхем.
- *Архитектура*: проектирование и реализация АЛУ/ЦПУ, такты и циклы, режимы адресации, логика выборки и выполнения, набор команд, ввод/вывод с привязкой к памяти.
- *Языки низкого уровня*: разработка и реализация простого машинного языка (двоичного и символического), наборы команд, программирование на языке ассемблера, ассемблеры.
- *Виртуальные машины*: автоматы на основе стека, арифметика стека, вызов и возврат функций, работа с рекурсией, разработка и реализация простого языка виртуальных машин.
- *Языки высокого уровня*: разработка и реализация простого объектно-ориентированного Java-подобного языка: абстрактные типы данных, классы, конструкторы, методы, правила разметки, синтаксис и семантика, ссылки.
- *Компиляторы*: лексический анализ, синтаксический разбор, таблицы символов, генерация кода, реализация массивов и объектов, двухуровневая компиляция.
- *Программирование*: реализация ассемблера, виртуальной машины и компилятора, следуя предоставленным API. Может быть выполнено на любом языке программирования.
- *Операционные системы*: разработка и реализация управления памятью, математические библиотеки, драйверы ввода/вывода,

обработка строк, текстовый вывод, графический вывод, поддержка языков высокого уровня.

- *Структуры данных и алгоритмы*: стеки, хеш-таблицы, списки, деревья, арифметические алгоритмы, геометрические алгоритмы, соображения по поводу времени исполнения.
- *Программная инженерия*: модульное проектирование, парадигма интерфейса/реализации, проектирование и документирование API, модульное (поблочное) тестирование, проактивное планирование тестирования, обеспечение качества, программирование на большом уровне (программирование крупных систем).

Уникальная особенность практической программы «От Nand до “Тетриса”» заключается в том, что все эти темы связно представлены и четко ориентированы на главную цель: создание современной компьютерной системы с нуля. Собственно, это и было нашим критерием отбора тем: минимальный набор знаний, необходимый для конструирования универсальной компьютерной системы, способной выполнять программы, написанные на объектно-ориентированном языке высокого уровня. Как оказалось, этот критический набор включает в себя большинство фундаментальных концепций и методов, а также некоторые из самых красивых идей прикладной информатики.

Курсы

Курсы «От Nand до “Тетриса”» обычно предлагаются как общие курсы для студентов и аспирантов, а также пользуются большой популярностью среди тех, кто учится самостоятельно. Курсы на основе этой книги, можно сказать, «перпендикулярны» типичному учебному плану по информатике и компьютерным наукам (*Computer Science*), их можно проходить на любой стадии обучения. Два естественных места в учебном плане — это CS-2, то есть вводный курс, но уже после основ программирования, и CS-99 — обобщающий курс в конце программы. Первый предполагает системно-ориентированное введение в прикладную информатику с заделом на будущее, а второй представляет

собой обобщающий блок, заполняющий пробелы, оставленные предыдущими курсами.

Еще один набирающий популярность вариант — это курс, объединяющий в одной структуре ключевые темы из традиционных курсов по архитектуре компьютера и курсов по компиляции. В любом случае, какова бы ни была обозначенная цель «От Nand до “Тетриса”», они могут иметь самые разные названия, в том числе «Элементы вычислительных систем», «Конструирование цифровых систем», «Организация компьютера», «Создаем компьютер» и, конечно же, «От Nand до “Тетриса”».

Книга и проекты имеют модульную структуру, начиная с самого основного разделения на часть I «Аппаратное обеспечение» и часть II «Программное обеспечение», каждая из которых состоит из шести глав и шести проектов. Хотя мы рекомендуем пройти весь курс, вполне возможно изучать каждую из двух частей отдельно. Книга и проекты могут поддерживать два независимых курса, каждый продолжительностью шесть-семь недель, то есть типичный семестровый курс или два семестровых курса — в зависимости от выбора тем и темпа обучения. Книга полностью самодостаточна: все необходимые знания для построения описанных в ней аппаратных и программных систем предоставляются в ее главах и проектах. Часть I «Аппаратное обеспечение» не требует предварительных знаний, что делает проекты 1–6 доступными для любого студента и самоучки. Часть II «Программное обеспечение» и проекты 7–12 требуют предварительного изучения программирования (на любом языке высокого уровня).

Курсы «От Nand до “Тетриса”» предназначены не только для тех, кто специализируется на информатике или «компьютерных науках». Они подходят для студентов любой специальности, желающих получить практические знания в областях аппаратных архитектур, операционных систем, компиляции и разработки программного обеспечения — и все это в рамках одного курса. И опять-таки единственным предварительным условием (для части II) является знание основ программирования. Действительно, многие студенты курса «От Nand до “Тетриса”» — это те, кто не выбирал в качестве своей основной специальности информатику, но прошел курс введения в информатику и пожелал узнать больше, не перегружая себя программой

из нескольких курсов. Довольно много среди учащихся и разработчиков программного обеспечения, пожелавших «спуститься пониже» и понять, как работают собственно технологии, чтобы повысить свое мастерство высокоуровневого программирования.

В связи с острой нехваткой разработчиков в индустрии аппаратного и программного обеспечения растет спрос на компактные и целенаправленные образовательные программы по прикладной информатике. Они часто принимают форму учебных лагерей программирования и онлайн-курсов, разрабатываемых для подготовки учащихся к рынку труда без прохождения полного курса и получения академической степени. Любая сколько-нибудь серьезная образовательная программа должна предлагать как минимум рабочие знания в области программирования, алгоритмов и систем. «От Nand до “Тетриса”» — это уникальная возможность охватить системные элементы таких программ в рамках одного курса. Кроме того, проекты «От Nand до “Тетриса”» — это привлекательное средство синтеза и практического применения знаний в области алгоритмов и программирования, полученных на других курсах.

Ресурсы

Все необходимые инструменты для создания описанных в книге аппаратных и программных систем свободно предоставляются в комплекте программного обеспечения «От Nand до “Тетриса”». В их число входят симулятор аппаратного обеспечения, эмулятор ЦПУ, виртуальная машина-эмулятор (все из открытых источников), учебные материалы и описанные в книге исполняемые версии ассемблера, виртуальной машины, компилятора и операционной системы. Кроме того, на сайте www.nand2tetris.org размещены все материалы проекта — около двухсот тестовых программ и скриптов, что позволяет постепенно разрабатывать и тестировать каждый из двенадцати проектов. Программные инструменты и материалы проекта можно использовать в исходном виде на любом компьютере под управлением Windows, Linux или macOS.

Структура

Часть I «Аппаратное обеспечение» охватывает главы 1–6. В главе 1 после введения в булеву алгебру описывается логический элемент (вентиль) Nand (И-НЕ), на основе которого строится набор других элементарных логических вентилей. В главе 2 описывается комбинационная логика и создание набора сумматоров, что подводит к конструированию АЛУ. В главе 3 описывается последовательностная (секвенциальная) логика и создание регистров и устройств памяти, что подводит к конструированию ОЗУ. В главе 4 обсуждается низкоуровневое программирование и разбирается машинный язык в его символической и двоичной формах. Глава 5 объединяет микросхемы, описанные в главах 1–3, в аппаратную архитектуру, способную выполнять программы, написанные на машинном языке, представленном в главе 4. В главе 6 обсуждается низкоуровневая трансляция программ, что завершается созданием ассемблера.

Часть II «Программное обеспечение» состоит из глав 7–12 и требует подготовки в области программирования (на любом языке) на уровне вводного курса информатики. В главах 7–8 разбираются стековые автоматы и описано построение JVM-подобной виртуальной машины.

В главе 9 описывается объектно-ориентированный Java-подобный язык высокого уровня. В главах 10–11 обсуждаются алгоритмы синтаксического анализа и генерации кода, а также описывается построение двухуровневого компилятора. В главе 12 представлены различные алгоритмы управления памятью, алгебраические и геометрические алгоритмы и описана реализация операционной системы, которая применяет эти алгоритмы на практике. Операционная система предназначена для устранения разрывов между языком высокого уровня, реализованным в части II, и аппаратной платформой, построенной в части I.

Книга основана на парадигме *«от абстракции к реализации»*. Каждая глава начинается с «Введения», в котором описываются соответствующие концепции и типовая аппаратная или программная система. Следующим разделом всегда следует «Спецификация», описывающая системную абстракцию, то есть различные операции, которые она должна тем или иным образом осуществлять. После описания

сути («что») каждая глава переходит к обсуждению реализации абстракции («как») в разделе «Реализация». Далее всегда следует раздел «Проект» с пошаговыми инструкциями, материалами для тестирования и программными инструментами для создания и модульного тестирования описанных в этой главе систем. Заключительный раздел «Перспектива» освещает заслуживающие внимания вопросы, оставшиеся за рамками главы.

Проекты

В книге описывается компьютерная система, которую можно создать *на самом деле*. Она предназначена для сборки, и она работает! Книга ориентирована на активных читателей, готовых поработать руками и создать компьютер с нуля. Потратив достаточно времени и сил, вы глубже поймете затронутые в книге темы и испытаете удовлетворение, не сравнимое с простым удовольствием от чтения.

Аппаратные устройства, созданные в проектах 1, 2, 3 и 5, реализуются с помощью простого языка описания аппаратуры (HDL) и моделируются на прилагаемом программном симуляторе аппаратного обеспечения, что в точности соответствует тому, как архитекторы аппаратуры работают в индустрии. Проекты 6, 7, 8, 10 и 11 (ассемблер, виртуальная машина I + II и компилятор I + II) могут быть написаны на любом языке программирования. Проект 4 пишется на языке ассемблера компьютера, а проекты 9 и 12 (простая компьютерная игра и базовая операционная система) пишутся на Jack — Java-подобном языке высокого уровня, для которого мы создаем компилятор в главах 10 и 11.

Всего получается двенадцать проектов. В среднем каждый проект соответствует недельному объему домашней работы типичного университетского курса. Проекты предназначены для самостоятельной работы, и их можно выполнять (или пропускать) в любом порядке. Полный курс «От Nand до “Тетриса”» подразумевает выполнение всех проектов в порядке их описания, но это только один из вариантов.

Возможно ли охватить столько материала в течение всего лишь одного семестра? Ответ — да, возможно, и это доказано на практике:

курсы «От Nand до “Тетриса”» длительностью в один семестр предлагают более 150 университетов. Студенты говорят об исключительной удовлетворенности этими программами, а открытые курсы дистанционного обучения «От Nand до “Тетриса”» регулярно попадают в верхние строчки рейтингов онлайн-курсов. Одна из причин, по которым учащиеся так активно реагируют на нашу методику, — это ее *сфокусированность*. За исключением очевидных случаев, мы не уделяем внимания оптимизации, оставляя этот важный предмет для других, более специфических курсов. Кроме того, мы исходим из предположения о безошибочном вводе данных. Это устраняет необходимость написания кода для обработки исключений и делает программные проекты значительно более сфокусированными и управляемыми. Работа с ошибочными входными данными, конечно, очень важна, но этот навык можно отточить и в других местах, например, во время работы над дополнительными проектами и на специальных курсах по программированию и проектированию программного обеспечения.

Второе издание

Хотя структура курса «От Nand до “Тетриса”» всегда строилась на двух основных темах, во втором издании она становится явной: теперь книга разделена на две отдельные и самостоятельные части — часть I «Аппаратное обеспечение» и часть II «Программное обеспечение». Каждая из них состоит из шести глав и шести проектов, начинаясь с заново написанного введения, излагающего основы главы каждой части. Важно отметить, что обе части независимы друг от друга. Таким образом, новая структура книги подходит для курсов, рассчитанных как на четверть, так и на семестр.

Помимо двух новых вводных глав, второе издание содержит четыре новых приложения. По просьбам многих слушателей в них в кратком концентрированном виде излагаются различные технические темы, отдельные аспекты которых в первом издании были разбросаны по главам. Еще одно новое приложение посвящено формальному доказательству того, что любую булеву функцию можно выразить

с помощью операторов Nand, что добавляет теоретическую перспективу прикладным проектам конструирования аппаратного обеспечения. Добавлено множество новых разделов, рисунков и примеров.

Все главы и материалы проекта были переписаны с упором на отделение абстракции от реализации, что служит основной темой практической программы «От Nand до “Тетриса”». Также мы потрудились, чтобы добавить примеры и разделы, отвечающие на тысячи вопросов, заданных на протяжении многих лет в разделах вопросов на форумах «От Nand до “Тетриса”».

Благодарности

Программные инструменты, дополняющие книгу, были разработаны нашими студентами из Междисциплинарного центра в Герцлии и Еврейского университета. Двумя главными архитекторами программного обеспечения были Ярон Украинаиц и Яннай Гончаровски, а разработчиками — Ифтах Иан Амит, Ассаф Гад, Галь Катцхендлер, Хадар Розен-Сиор и Нир Розен. Над другими аспектами инструментов работали Орен Баранес, Орен Коэн, Джонатан Гросс, Голан Параши и Ури Зейра. Работать с этими студентами-разработчиками было очень приятно, и мы гордимся тем, что сыграли свою роль в их образовании.

Мы также благодарим наших ассистентов-преподавателей Муавию Акаша, Филипа Хендрикса, Эйтана Лифшица, Рана Навока и Давида Рабиновица, которые помогали вести ранние версии курса, на основе которого была написана эта книга. Тал Ачитув, Йонг Бакос, Тали Гутман и Михаэль Шредер оказали большую помощь в работе над различными аспектами материалов курса, а Арье Шналл, Томаш Ружаньский и Рудольф Адамкович предложили тщательные редакционные правки. Особенно познавательными были комментарии Рудольфа, за что мы ему очень благодарны.

В работе над практической программой «От Nand до “Тетриса”» приняли участие многие люди по всему миру, и мы не можем поблагодарить их всех по отдельности. Но мы сделаем одно исключение. Настоящим ангелом-хранителем учеников «От Nand до “Тетриса”» стал

Марк Армбруст, инженер-программист из Колорадо. Вызвавшись управлять нашим глобальным форумом вопросов и ответов, Марк отвечал на многочисленные вопросы с большим терпением, и его ответы отличались изящным стилем. Он никогда не описывал решения напрямую; он скорее подталкивал учащихся в верном направлении и помогал им найти решение самостоятельно. Благодаря этому Марк завоевал уважение и восхищение многочисленных учеников по всему миру. Работая на передовой «От Nand до “Тетриса”» более десяти лет, Марк опубликовал 2 607 сообщений, обнаружил десятки ошибок, написал корректирующие скрипты и исправления. Делая все это в дополнение к своей обычной работе, Марк стал настоящей опорой сообщества «От Nand до “Тетриса”», а сообщество стало его вторым домом. В марте 2019 года, после нескольких месяцев борьбы с болезнью сердца, Марк скончался. Во время госпитализации он ежедневно получал сотни электронных писем от студентов «От Nand до “Тетриса”». Юноши и девушки со всего мира благодарили Марка за его безграничную щедрость и восхищались тем влиянием, которое он оказал на их жизнь.

В последние годы образование в области компьютерных наук стало мощным фактором личностного роста и экономической мобильности. Оглядываясь назад, мы считаем, что нам повезло и что мы не зря решили с самого начала сделать все наши учебные ресурсы свободно доступными и с открытым исходным кодом. Проще говоря, любой желающий может не только изучать, но и преподавать курс «От Nand до “Тетриса”» без каких бы то ни было ограничений. Все, что для этого нужно, — зайти на наш сайт и загрузить материалы, при условии, что вы работаете в некоммерческой сфере. Так курс «От Nand до “Тетриса”» превратился в легкодоступное для всех средство распространения высококачественного образования в области информатики. В результате возникла обширная образовательная экосистема, подпитываемая бесконечными запасами доброй воли. Мы благодарим многих людей по всему миру, которые помогли нам воплотить в жизнь эти мечты.

I. Аппаратное обеспечение

Настоящее открытие — это не поиск новых земель, это взгляд на мир новыми глазами.

— *Марсель Пруст (1871–1922)*

Эта книга — путешествие в мир открытий. Вам предстоит узнать три вещи: как работают компьютерные системы, как разбивать сложные проблемы на управляемые модули и как создавать крупномасштабные аппаратные и программные системы. Вы совершите практическое путешествие, в ходе которого создадите полную и работающую компьютерную систему с нуля. В ходе работы в качестве «побочного эффекта» вы усвоите нечто, что будет гораздо важнее самого компьютера. Как сказал психолог Карл Роджерс: «Единственный вид обучения, который существенно влияет на поведение, — это самостоятельное открытие или самостоятельно усвоенная на своем опыте истина». Эта вводная глава описывает некоторые открытия, истины и опыт, которые ждут нас впереди.

Привет, нижний мир

Если у вас есть некоторый опыт программирования, то в самом начале обучения вы наверняка сталкивались с чем-то вроде приведенной ниже программы. Но даже если и нет, то вы все равно можете догадаться, что она делает: выводит текст `Hello World` и останавливается.

Данная программа написана на языке Jack — простом, похожем на Java языке высокого уровня:

```
// Первый пример в программировании 101
class Main {
    function void main() {
        do Output.printString("Hello World");
        return;
    }
}
```

Тривиальные программы вроде **Hello World** обманчиво просты. Задумывались ли вы когда-нибудь о том, что *на самом деле* происходит, когда на компьютере запускаются эти программы? Заглянем, так сказать, «под капот». Для начала отметим, что, собственно, программа — это не более чем последовательность простых символов, хранящихся в текстовом файле. Эта абстракция — полная загадка для компьютера, который понимает только команды (называемые также *инструкциями*), написанные на машинном языке. Таким образом, если мы хотим выполнить данную программу, то первое, что необходимо сделать, — это разобрать строку символов, из которых состоит код высокого уровня, и раскрыть его семантику, то есть понять, что программа хочет сделать. Затем необходимо сгенерировать код низкого уровня, который выразит эту семантику по-своему, на машинном языке данного компьютера. Результатом такого сложного процесса перевода, известного как *компиляция*, станет исполняемая последовательность инструкций машинного языка.

Конечно, машинный язык — это тоже абстракция, набор двоичных символов с заранее установленными значениями. Для того чтобы эта абстракция стала конкретной, она должна быть реализована в некоторой *аппаратной архитектуре*. А эта архитектура, в свою очередь, реализуется определенным набором микросхем — регистров, блоков памяти, сумматоров и т. д. Каждое из этих аппаратных устройств построено из элементарных единиц — *логических элементов* (вентилей) более низкого уровня. Эти логические вентили

можно создать из таких примитивных (самых простых) логических элементов, как *Nand* и *Nor*. Примитивные логические элементы находятся очень низко в иерархии, но тоже состоят из нескольких переключающих устройств, обычно реализуемых транзисторами. А каждый транзистор состоит из... Впрочем, углубляться дальше мы не будем, потому что на этом компьютерная наука заканчивается и начинается физика.

Вы можете подумать: «Ну, на моем компьютере скомпилировать и запустить программу гораздо проще — достаточно только нажать на эту иконку или напечатать команду!» И действительно, современная компьютерная система похожа на погруженный в воду айсберг: большинство людей видят только его верхушку, и их знания о вычислительных системах отрывочны и поверхностны. Но если вы хотите исследовать, что находится под поверхностью, то вам необычайно повезло! Там, внизу, находится удивительный мир, состоящий из самых красивых материалов во всей компьютерной науке. И именно глубокое знание этого скрытого мира отделяет начинающих программистов от искушенных разработчиков — людей, которые умеют создавать сложные аппаратные и программные технологии. Лучший же способ понять, как они работают, — мы имеем в виду понять их до мозга костей — это построить полную компьютерную систему с нуля.

От Nand до «Тетриса»

Допустим, мы решили создать компьютерную систему с нуля, но какой именно компьютер мы должны собрать? Оказывается, любой компьютер общего назначения, то есть любой ПК, смартфон или сервер, представляет собой машину «От Nand до “Тетриса”». Во-первых, в основе всех компьютеров лежат базовые логические элементы (вентили), из которых в индустрии наиболее широко используются элементы Nand (что такое вентиль Nand, мы объясним далее, в главе 1). Во-вторых, любой компьютер общего назначения можно запрограммировать на запуск игры «Тетрис», а также любой другой программы,

идея которой придет вам в голову. Таким образом, ни в Nand, ни в «Тетрисе» нет ничего уникального. Уникальность и волшебство предстоящего вам путешествия заключается именно в предложениях «от» и «до» — книга предлагает вам пройти весь путь от элементарных устройств-переключателей до машины, на которой можно обрабатывать текст, графику, анимацию, музыку, видео, заниматься анализом, запускать симуляции, разрабатывать искусственный интеллект и делать все, что мы привыкли ожидать от универсальных компьютеров. Поэтому не имеет значения, какую именно аппаратную платформу и какую иерархическую структуру программной системы мы выберем, если они основываются на идеях и методах, характерных для всех существующих вычислительных систем.

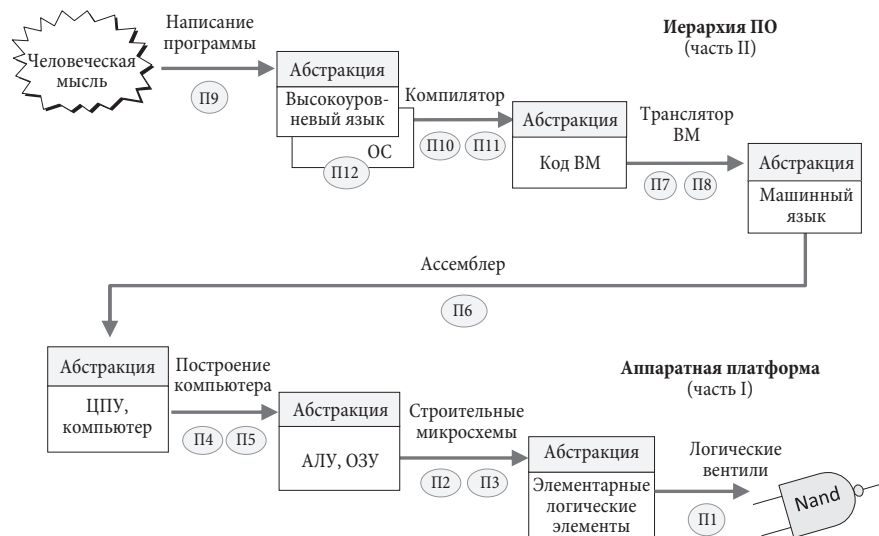


Иллюстрация I.1. Основные модули типичной компьютерной системы, состоящей из аппаратной платформы и иерархической структуры программной системы. Каждый модуль имеет *абстрактное представление* (также называемое *интерфейсом модуля*) и *реализацию*. Направленные вправо стрелки означают, что каждый модуль реализуется с использованием абстрактных блоков, находящихся уровнем ниже. Каждый кружок обозначает проект программы «От Nand до «Тетриса»» и главу книги — всего двенадцать проектов и соответствующих глав.

На рисунке I.1 показаны основные этапы пути от Nand до «Тетриса». Если рассматривать его с нижнего уровня, то видно, что любой универсальный компьютер имеет архитектуру, включающую в себя АЛУ (арифметико-логическое устройство) и ОЗУ (оперативное запоминающее устройство, или «память с произвольным доступом»). Все устройства АЛУ и ОЗУ состоят из элементарных логических вентилей. И, к удивлению и к счастью, как мы вскоре увидим, любые логические вентили можно создать только на основе вентилей Nand. Что касается иерархии программного обеспечения (иерархической структуры программной системы), то все языки высокого уровня опираются на набор трансляторов (компилятор/интерпретатор, виртуальная машина, ассемблер), преобразующих высокоуровневый код в команды машинного уровня. Некоторые языки высокого уровня интерпретируются, а не компилируются, некоторые не используют виртуальную машину, но в целом картина примерно одна и та же. Это наблюдение — проявление фундаментального принципа информатики, известного как *гипотеза (тезис) Черча — Тьюринга*: в основе своей все компьютеры эквивалентны.

Мы обращаем внимание на эти наблюдения, чтобы подчеркнуть общий характер нашего подхода: проблемы, идеи, советы, приемы, технические методы и терминология, с которыми вы столкнетесь в этой книге, точно такие же, какими оперируют и какие используют практикующие инженеры по аппаратному и программному обеспечению. В этом отношении «От Nand до “Тетриса”» служит своего рода инициацией: в случае удачного прохождения курса вы получите отличную основу для того, чтобы стать профессионалом в области компьютерных технологий.

Итак, на какую же конкретную аппаратную платформу и на какой конкретный язык высокого уровня лучше практически ориентироваться в курсе «От Nand до “Тетриса”»? Один из вариантов — собрать компьютер, похожий на выпускаемые промышленностью и широко используемые модели, и написать компилятор для популярного языка высокого уровня. Мы отказались от этого варианта по трем причинам. Во-первых, компьютерные модели приходят и уходят, а популярные языки программирования уступают место новым. Поэтому мы не хотели привязываться к какой-то конкретной конфигурации

аппаратного и программного обеспечения. Во-вторых, компьютеры и языки, используемые на практике, содержат множество деталей, которые не имеют особой обучающей ценности, но на реализацию которых уходит целая вечность. Наконец, нам нужны были аппаратная платформа и программная иерархия, которые можно было бы легко контролировать, понимать и расширять. Эти соображения привели к разработке компьютерной платформы Наск, о создании которой говорится в первой части книги, и высокоуровневого языка Jack, который описан во второй части.

Обычно компьютерные системы описываются *«сверху вниз»*, то есть посредством объяснения того, как высокоуровневые абстракции сводятся к более простым или реализуются более простыми абстракциями. Например, можно описать, как двоичные машинные команды, выполняемые на архитектуре компьютера, разбиваются на микрокоды, проходящие по проводам архитектуры и в конечном счете управляющие микросхемами АЛУ и ОЗУ нижнего уровня. В качестве альтернативы можно предложить описание *«снизу вверх»*, например, показать, как микросхемы АЛУ и ОЗУ специально проектируются для выполнения микрокодов, которые в своей совокупности образуют двоичные машинные команды. Оба подхода — *«сверху вниз»* и *«снизу вверх»* — познавательны, и каждый из них позволяет по-разному взглянуть на систему, которую мы собираемся построить.

Направление стрелок на иллюстрации I.1 предполагает ориентацию *«сверху вниз»*. Для любой заданной пары модулей существует стрелка, направленная вправо и соединяющая модуль более высокого уровня с модулем более низкого уровня. Смысл этой стрелки однозначен: она подразумевает, что модуль более высокого уровня реализуется с помощью абстрактных строительных блоков уровнем ниже. Например, программа высокого уровня реализуется посредством перевода каждого высказывания (команды) высокого уровня в набор абстрактных команд виртуальной машины (ВМ). А каждая команда ВМ, в свою очередь, переводится далее в набор абстрактных команд (инструкций) машинного языка. И так далее. Различие между абстракцией и реализацией играет важную роль в проектировании систем, о чем мы сейчас и поговорим.

Абстракция и реализация

Вы можете задаться вопросом, как же это возможно — собрать полную компьютерную систему с нуля, начиная с элементарных логических вентилей. Наверняка это очень сложное и грандиозное предприятие! Но с любой сложностью можно справиться, разбивая систему на *модули*. Каждый модуль описывается отдельно, в отдельной главе, и собирается отдельно — в самостоятельном проекте. Далее вы можете задаться вопросом, как же возможно описать и сконструировать эти модули по отдельности, ведь, по всей видимости, все они как-то взаимосвязаны. Но, как показано на протяжении всей этой книги, хорошая модульная конструкция подразумевает следующую идею: работать над отдельными модулями можно независимо, полностью игнорируя остальные части системы. В действительности, если система хорошо спроектирована, можно собирать данные модули в любом порядке и даже параллельно друг другу, работая в коллективе.

Принцип «разделяй и властвуй», то есть когнитивная способность человека разбивать сложную систему на управляемые модули, поддерживается еще одним нашим когнитивным даром: способностью различать *абстракцию* и *реализацию* каждого модуля. В компьютерной науке мы воспринимаем эти слова очень конкретно: абстракция описывает то, *что* делает модуль, а реализация — *как* он это делает. Опираясь на данное различие, можно сформулировать самое важное правило системной инженерии: при использовании модуля — *любого модуля* — в качестве строительного блока следует сосредотачиваться исключительно на абстракции модуля, полностью игнорируя подробности его реализации.

Для примера сосредоточимся на нижнем слое иллюстрации I.1, начиная с уровня «архитектура компьютера». Как видно на рисунке, при реализации данной архитектуры используется несколько строительных блоков нижнего уровня, включая оперативную память (ОЗУ, оперативное запоминающее устройство, или «память с произвольным доступом»). ОЗУ — удивительная вещь. Оно может содержать миллиарды регистров, и к любому из них возможно получить прямой

доступ, причем практически мгновенно. Иллюстрация I.1 говорит нам о том, что архитектор компьютера должен использовать это устройство прямого доступа абстрактно, не обращая внимания на то, как оно реализовано на самом деле. Все, что связано с его созданием: умственные и физические усилия, решения возникающих в ходе его создания проблем, конкретные принципы работы, то есть вся «магия» ОЗУ, его «как» — все это при сборке компьютера следует полностью проигнорировать, поскольку данная информация не имеет значения в контексте *использования* ОЗУ ради достижения желаемого эффекта.

Спустившись на один уровень вниз на иллюстрации I.1, мы оказываемся в положении, когда нужно собрать микросхему оперативной памяти. Как нам следует поступить? Следуя по направленной вправо стрелке, мы видим, что реализация ОЗУ основывается на элементарных логических вентилях и микросхемах более низкого уровня. В частности, возможности хранения и прямого доступа к ОЗУ реализуются с помощью *регистров* и *мультиплексоров* соответственно. И снова срабатывает тот же принцип «абстракции — реализации»: мы пользуемся данными микросхемами как абстрактными строительными блоками, сосредотачиваясь на их интерфейсах и не заботясь об их *реализации*. И так далее — вплоть до уровня Nand.

Итак, говоря вкратце, всякий раз, когда реализация подразумевает использование аппаратного или программного модуля более низкого уровня, нужно относиться к этому модулю как к готовой абстракции, как к своего рода «черному ящику»: все, что нужно, — это документация интерфейса модуля, описывающая, *что* он может делать, и больше ничего. Не нужно обращать никакого внимания на то, *как* модуль выполняет то, что заявлено в его интерфейсе. Эта парадигма «от абстракции к реализации» помогает разработчикам справляться со значительной сложностью и сохранять здравомыслие. Разделяя чрезвычайно сложную систему на четко определенные модули, мы упрощаем задачу по реализации, работая над отдельными частями, которыми относительно легко манипулировать, и заодно упрощаем задачу по локализации и исправлению ошибок. Это самый важный принцип проектирования в работе по созданию аппаратного и программного обеспечения.

Не стоит лишний раз подчеркивать, что все в этой истории зависит от сложного искусства *модульного проектирования*: человеческой способности разделять поставленную задачу на элегантный набор четко определенных модулей, каждый из которых имеет ясный интерфейс, предполагающий свой отдельный этап работы, и каждый из которых тестируется с помощью отдельной независимой программы модульного тестирования. И действительно, модульное проектирование — это костяк прикладной информатики: каждый системный архитектор регулярно формулирует абстракции, иногда называемые *модулями* или *интерфейсами*, а затем реализует их или просит об этом других специалистов. Абстракции часто разрабатываются слой за слоем, что приводит к появлению все более высоких уровней функциональности. Если системный архитектор разрабатывает хороший набор модулей, работа по реализации протекает быстро и безупречно; если дизайн модулей небрежен — реализация обречена.

Модульное проектирование — это искусство, которому можно обучиться, рассматривая и реализуя множество хорошо продуманных абстракций. Именно это вам предстоит освоить в практическом курсе «От Nand до “Тетриса”»: вы научитесь ценить элегантность и функциональность сотен аппаратных и программных абстракций. Вам расскажут, как реализовывать каждую из этих абстракций шаг за шагом и создавать все более крупные функциональные фрагменты. По мере продвижения в данном направлении, переходя от одной главы к другой, вы будете с удовольствием оглядываться на проделанный путь и оценивать компьютерную систему, постепенно обретающую форму в результате ваших усилий.

Методология

Путешествие «От Nand до “Тетриса”» предполагает создание аппаратной платформы и иерархии программного обеспечения. Аппаратная платформа основана на наборе из примерно тридцати логических вентилях и микросхем, о построении которых говорится в первой части книги. Каждый из этих вентилях и каждая микросхема, включая

самую верхнюю компьютерную архитектуру, будут созданы с использованием языка описания аппаратуры (HDL, *Hardware Description Language*). Используемый нами HDL задокументирован в приложении 2, и его можно изучить примерно за час. Правильность работы своих HDL-программ вы будете проверять с помощью программного симулятора аппаратуры, работающего на вашем персональном компьютере. Именно так на практике и работают инженеры по аппаратному обеспечению: они создают и тестируют микросхемы с помощью программных симуляторов. Если смоделированные характеристики микросхем их устраивают, они отправляют свои спецификации (HDL-программы) в компанию, занимающуюся производством микросхем. После оптимизации HDL-программы становятся командами для роботизированных манипуляторов, собирающих аппаратное обеспечение в физической форме (в виде кремниевых «чипов»).

Во второй части, двигаясь дальше по пути «От Nand до “Тетриса”», мы создадим «программный костяк», включающий ассемблер, виртуальную машину и компилятор. Такие программы можно реализовать на любом языке программирования высокого уровня. Кроме того, мы создадим базовую операционную систему, написанную на языке Jack.

Возможно, вы сомневаетесь в том, что такие амбициозные проекты можно реализовать в рамках одного курса или одной книги. Что ж, помимо модульной конструкции у нас в запасе имеется еще один тайный ингредиент — снижение неопределенности проектирования до абсолютного минимума. Для каждого проекта мы предоставляем четко продуманные планы и инструменты, включая подробные API, базовые программы, сценарии тестирования и поэтапное руководство по реализации.

Все программные инструменты, необходимые для выполнения проектов 1–12, доступны в программном пакете «От Nand до “Тетриса”», который можно бесплатно загрузить с сайта www.nand2tetris.org. В него входят симулятор аппаратного обеспечения, эмулятор ЦПУ, эмулятор виртуальной машины, а также исполняемые версии аппаратных чипов, ассемблера, компилятора и ОС. Как только вы загрузите пакет программ на свой компьютер, данные инструменты всегда будут у вас под рукой.

Путь, который вас ожидает

Путь «От Nand до “Тетриса”» включает в себя двенадцать проектов по созданию аппаратного и программного обеспечения. Общее направление этих проектов, как и список тем книги, предполагает путешествие снизу вверх: мы начинаем с элементарных логических вентилей и движемся по восходящей к высокоуровневому объектно-ориентированному языку программирования. В то же время развитие в рамках каждого отдельного проекта направлено сверху вниз. В частности, каждый раз, когда мы знакомим читателя или студента с аппаратным или программным модулем, то всегда начинаем с абстрактного описания того, *что* этот модуль должен делать и зачем он нужен. Осознав абстракцию модуля (что само по себе уже целый богатый мир), вы переходите к его реализации с использованием абстрактных строительных блоков с нижнего уровня.

Итак, рассмотрим же наконец грандиозный план первой части нашей экскурсии. В главе 1 мы начнем с одного логического вентиля Nand (И-НЕ) и построим на его основе набор элементарных и часто используемых логических вентилей, таких как And (И), Or (ИЛИ), Xor и т. д. В главах 2 и 3 используем эти строительные блоки для создания арифметико-логического устройства и устройств памяти соответственно. В главе 4 приостанавливаем наше путешествие по созданию аппаратного обеспечения и описываем низкоуровневый машинный язык в его символической и двоичной формах. В главе 5 используем построенные ранее АЛУ и устройства памяти для создания центрального процессора (ЦПУ) и памяти с произвольным доступом (ОЗУ). Затем эти устройства будут интегрированы в аппаратную платформу, способную выполнять программы, написанные на машинном языке, представленном в главе 4. В главе 6 мы опишем и сконструируем ассемблер, представляющий собой программу, переводящую низкоуровневые программы, написанные на символьном машинном языке, в исполняемый двоичный код. На этом построение аппаратной платформы будет завершено. Данная платформа станет отправной точкой для второй части книги, в которой мы дополним базовое аппаратное

обеспечение современной иерархией ПО, состоящей из виртуальной машины, компилятора и операционной системы.

Надеемся, нам удалось рассказать о том, что ожидает впереди, и вам уже не терпится отправиться в это грандиозное путешествие в поисках чудесных открытий. Итак, если вы готовы и настроены на рабочий лад, начинаем обратный отсчет: 1, 0 — поехали!

1. Булева логика

Такие простые вещи, а мы делаем из них нечто настолько сложное, что оно побеждает нас. Почти.

— Джон Эшбери (1927–2017)

В основе каждого цифрового устройства — будь то персональный компьютер, сотовый телефон или сетевой маршрутизатор — лежит набор микросхем, предназначенных для хранения и обработки двоичной информации. Хотя эти микросхемы бывают разных форм и видов, все они состоят из одних и тех же строительных блоков: элементарных *логических вентилях* (базовых логических элементов). Физически реализовать их можно по-разному, с помощью различных аппаратных технологий, но их логическое поведение, или абстракция, одно и то же во всех реализациях.

В этой главе мы начнем с одного примитивного логического вентиля Nand (И-НЕ) и на его основе построим все остальные логические вентили, которые нам понадобятся. В частности, вентили Not (НЕ), And (И), Or (ИЛИ) и Xor, а также два вентили под названиями «мультиплексор» и «демультиплексор» (функции их всех описаны ниже). Поскольку наш целевой компьютер будет рассчитан на работу с 16-битными значениями, мы также построим 16-битные версии основных вентилях — такие как Not16, And16 и т. д. В результате получим довольно стандартный набор логических вентилях, который в дальнейшем будет использован для построения микросхем обработки информации и памяти нашего компьютера. Это будет сделано в главах 2 и 3 соответственно.

Глава начинается с описания минимального набора теоретических понятий и практических инструментов, необходимых для проектирования и реализации логических вентилей. В частности, мы представляем булеву алгебру и булевы функции и показываем, как можно реализовать булевы функции с помощью логических вентилей. Затем мы описываем, как логические вентили можно реализовать с помощью языка описания аппаратуры (HDL) и как эти конструкции тестируются с помощью симуляторов аппаратного обеспечения. Это введение важно для всей первой части книги, поскольку булева алгебра и HDL используются в каждой из последующих глав и проектов по аппаратному обеспечению.

1.1. Булева алгебра

Булева алгебра имеет дело с двоичными значениями, которые обычно передаются как «ложь/истина» (true/false), 1/0, «да/нуль», «вкл/выкл» и т. д. Мы будем использовать значения «1» и «0». Булева функция — это функция, которая принимает на входе двоичное значение и выдает («возвращает») двоичное значение. Поскольку работа компьютерных устройств подразумевает обработку двоичных данных, булевы функции играют центральную роль в спецификации, анализе и оптимизации аппаратных архитектур.

Булевы операторы. На иллюстрации 1.1 показаны три часто используемые булевы функции, также известные под названием «булевы операторы». Эти функции называются And (И), Or (ИЛИ) и Not (НЕ); их также записывают с помощью обозначений $x \cdot y$, $x + y$ и $\bar{x}$ или $x \wedge y$, $x \vee y$ и x соответственно. На иллюстрации 1.2 показаны все возможные булевы функции, которые могут быть определены над двумя переменными, а также их самые распространенные названия. Эти функции были построены систематически посредством перебора всех возможных комбинаций значений двоичных переменных. Каждый оператор имеет условное название, описывающее его основную семантику. Например, название оператора Nand — это сокращение от «Not-And»

(«не И», или «И-НЕ»), потому что функция Nand (x, y) эквивалентна функции Not (And (x, y)). Оператор Xor — это сокращение от выражения «*exceptive or*» («исключающее ИЛИ»), он принимает значение 1, если только одна из двух переменных равна 1. Вентиль Nor получил свое название от Not-Or («не Или», «ИЛИ-НЕ»). Впрочем, точные названия всех этих вентилях не так уж важны.

x	y	x And y	x	y	x Or y	x	Not x
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Иллюстрация 1.1. Три элементарные булевы функции.

Тождественный 0	0	0 0 0 0
And (И)	$x \cdot y$	0 0 0 1
And Not y (x И НЕ y)	$x \cdot \bar{y}$	0 0 1 0
x	x	0 0 1 1
Not x And y (НЕ x И y)	$\bar{x} \cdot y$	0 1 0 0
y	y	0 1 0 1
Xor («исключающее ИЛИ»)	$x \cdot \bar{y} + \bar{x} \cdot y$	0 1 1 0
Or (ИЛИ)	$x + y$	0 1 1 1
Nor (ИЛИ-НЕ)	$\overline{x + y}$	1 0 0 0
Эквиваленция	$x \cdot y + \bar{x} \cdot \bar{y}$	1 0 0 1
Not y (НЕ y)	$\bar{y}$	1 0 1 0
If y then x (Если y , то x)	$x + \bar{y}$	1 0 1 1
Not x (НЕ x)	$\bar{x}$	1 1 0 0
If x then y (Если x , то y), импликация	$\bar{x} + y$	1 1 0 1
Nand (И-НЕ)	$\overline{x \cdot y}$	1 1 1 0
Тождественная 1	1	1 1 1 1

Иллюстрация 1.2. Все булевы функции двух двоичных переменных. В общем случае количество булевых функций, задаваемых n двоичными переменными (здесь $n = 2$), равно 2^{2^n} (а это очень много булевых функций).

Иллюстрация 1.2 заставляет задуматься: что делает операторы And (И), Or (ИЛИ) и Not (НЕ) более интересными или привилегированными, чем любое другое подмножество булевых операторов? Краткий ответ заключается в том, что в операторах And, Or и Not нет ничего особенного. Более же глубокий ответ значит, что для выражения любой булевой функции можно использовать различные подмножества логических операторов и одним из таких подмножеств является подмножество {And, Or, Not}. Если это вам показалось интересным, то следующее утверждение покажется еще более захватывающим: любой из трех данных основных операторов можно выразить с помощью одного лишь оператора Nand. Вот это уже впечатляет! Из этого следует, что любую булеву функцию можно реализовать с помощью одного лишь вентиля Nand. Доказательство такого замечательного утверждения приводится в необязательном для чтения приложении 1.

Булевы функции

Каждую булеву функцию можно задать с помощью двух альтернативных представлений. Во-первых, с помощью *таблицы истинности*, как показано на иллюстрации 1.3. Для каждого из 2^n возможных кортежей значений переменных $v_1, \dots, v_n$ (здесь $n = 3$) в таблице указано значение $f(v_1, \dots, v_n)$. В дополнение к этому определению, основанному на значениях данных, булевы функции можно задавать с помощью булевых выражений, например: $f(x, y, z) = (x \text{ Or } y) \text{ And Not}(z)$.

Как же проверить, что данное булево выражение эквивалентно данной таблице истинности? В качестве примера воспользуемся иллюстрацией 1.3. Начнем с первого ряда, соответствующего значению $f(0, 0, 0)$, то есть $(0 \text{ ИЛИ } 0) \text{ И } \text{НЕ}(0)$. Это выражение действительно равно 0, как и указано в таблице истинности. Пока что верно. Аналогичный тест на эквивалентность можно применить к каждой строке таблицы, но это довольно утомительное занятие. Вместо трудоемкого метода доказательства снизу вверх можно доказать эквивалентность сверху вниз, проанализировав булево выражение $(x \text{ Or } y) \text{ And Not}(z)$ в целом: $(x \text{ ИЛИ } y) \text{ И } \text{НЕ}(z)$. Обратив внимание на часть слева от оператора И, мы видим, что данное выражение $(x \text{ ИЛИ } y)$ равно 1 только тогда,

когда $x = 1$ или $y = 1$. Рассмотрев часть справа от оператора И, мы видим, что это выражение равно 1 только при $z = 0$. Сопоставив эти два наблюдения, приходим к выводу, что выражение в целом равно 1 только в случае $((x = 1) \text{ ИЛИ } (y = 1)) \text{ И } (z = 0)$. Это условие соблюдается только в строках 3, 5 и 7 таблицы истинности, и действительно только эти строки содержат единицу в крайнем правом столбце.

x	y	z	$f(x, y, z) = (x \text{ Or } y) \text{ And Not}(z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Иллюстрация 1.3. Таблица истинности и функциональный способ задания булевой функции (пример).

Таблицы истинности и булевы выражения

Если имеется некая булева функция от n переменных, представленная булевым выражением, то для нее всегда можно построить таблицу истинности. Для этого нужно всего лишь вычислить значение функции для каждого набора переменных, то есть для каждой строки таблицы. Такая конструкция очевидна, хотя и трудоемка с практической точки зрения. В то же время совсем не очевидна другая, противоположная конструкция: можно ли на основании некоей таблицы истинности построить булево выражение для функции, лежащей в основе данной конкретной таблицы? Ответ — да, что весьма любопытно. Доказательство можно найти в приложении 1.

Когда речь заходит о создании компьютеров и таблицы истинности, булевы выражения и возможность построения одних на основе других приобретают очень большое значение. Предположим для примера, что нам поручили создать некую аппаратуру для секвенирования данных

ДНК и что наш специалист-биолог хочет описать логику секвенирования с помощью некоей таблицы истинности. Наша задача — реализовать логику в аппаратуре. Взяв за отправную точку данные таблицы истинности, мы можем сформулировать на ее основе булево выражение, представляющее нужную функцию. Упростив выражение с помощью булевой алгебры, мы сможем приступить к реализации аппаратной платформы с помощью логических вентилях, что и сделаем далее в этой главе. Подводя итог, можно сказать, что таблица истинности часто бывает удобным средством для описания некоторых природных состояний, а булевы выражения — прекрасное средство формализации данного описания и перевода его на язык кремниевых чипов. Возможность перехода от одного представления к другому — один из важнейших практических методов при проектировании аппаратуры.

Мимоходом мы заметили, что, хотя представление булевой функции в виде таблицы истинности уникально, каждая булева функция может быть представлена множеством различных, но эквивалентных булевых выражений. Некоторые из них будут короче, и с ними будет легче работать. Например, сложное выражение $(\text{Not}(x \text{ And } y) \text{ And } (\text{Not}(x) \text{ Or } y) \text{ And } (\text{Not}(y) \text{ Or } y))$ эквивалентно простому выражению $\text{Not}(x)$. Понятно, что возможность упростить булево выражение — это первый шаг к аппаратной оптимизации. Такое упрощение производится с помощью булевой алгебры и здравого смысла, как и показано в приложении 1.

1.2. Логические вентили

Логический *вентиль* — это физическое устройство, реализующее простую булеву функцию. Хотя в настоящее время в большинстве цифровых компьютеров для реализации вентилях и представления двоичных данных используются электрические компоненты, для этого можно использовать любые альтернативные технологии, позволяющие передавать и обрабатывать данные. И действительно, за долгие годы было создано множество различных аппаратных реализаций

булевых функций, включая магнитные, оптические, биологические, гидравлические, пневматические и квантовые механизмы (некоторые из которых представляют собой любопытные любительские поделки, например, на основе костяшек домино). В настоящее время вентили обычно реализуются в виде транзисторных элементов, вытравленных в кристалле кремния и упакованных в виде «чипа», то есть микросхемы. В проекте «От Nand до “Тетриса”» мы используем слова «чип» («микросхема») и «вентиль» как взаимозаменяемые, но склоняемся к использованию последнего для обозначения простых примеров первых.

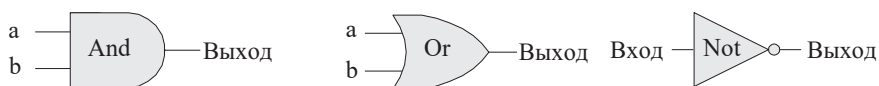


Иллюстрация 1.4. Стандартные графические обозначения трех элементарных логических вентилях.

Исходя из того, что логические вентили можно создавать на основе различных технологий, а также того, что булеву алгебру можно использовать для абстрактного анализа работы логических элементов, мы можем сделать крайне важный вывод. По сути, это означает, что специалистов в области компьютерных технологий не должны беспокоить такие физические артефакты, как электричество, цепи, переключатели, реле и источники питания. Специалисты по компьютерам довольствуются абстрактными понятиями булевой алгебры и логики вентилях, уповая на то, что кто-то другой — например, физики и инженеры-электрики — придумает, как реализовать их в аппаратуре. Следовательно, примитивные вентили, подобные тем, что показаны на иллюстрации 1.4, можно рассматривать как устройства типа «черный ящик», реализующие элементарные логические операции тем или иным способом — нас не волнует, каким именно. Использование булевой алгебры для анализа абстрактного поведения логических вентилях описал в 1937 году Клод Шеннон, защитивший на основе данных рассуждений магистерскую диссертацию, которую иногда называют самой важной диссертацией в области информатики.

Примитивные и составные вентили

Поскольку все логические вентили имеют одинаковые типы входных и выходных данных (0 и 1), их можно комбинировать, создавая составные вентили произвольной сложности. Например, предположим, что нас попросили реализовать тернарную (с тремя переменными) булеву функцию And (a, b, c), которая возвращает 1, если каждая переменная равна 1, в противном же случае результат должен быть равен 0. Прибегнув к булевой алгебре, мы можем начать с наблюдения, что $a \cdot b \cdot c = (a \cdot b) \cdot c$, или, если записать это в префиксной нотации, $\text{And}(a, b, c) = \text{And}(\text{And}(a, b), c)$. Далее мы можем использовать этот результат для построения составных вентилях, изображенных на иллюстрации 1.5.

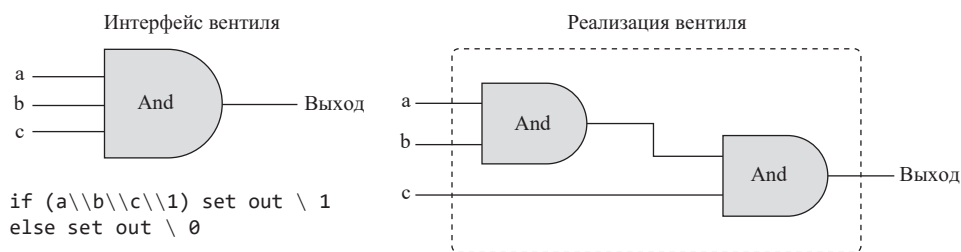


Иллюстрация 1.5. Составная реализация логического вентиля And с тремя входами. Пунктирный прямоугольник обозначает границы интерфейса вентиля.

Как видно, любой логический вентиль можно рассматривать с двух разных точек зрения: внутренней и внешней. В правой части иллюстрации 1.5 показана внутренняя архитектура, или *реализация*, а в левой показан *интерфейс* вентиля, а именно его входные и выходные контакты и логика поведения, важная для внешнего окружения. Внутреннее представление имеет значение только для разработчика вентиля, в то время как внешнее представление — необходимый уровень детализации для проектировщиков, которые собираются использовать вентиль как абстрактный, готовый компонент, не задумываясь о его реализации.

Рассмотрим еще один пример логической конструкции: Хор. По своему определению функция $\text{Xor}(a, b)$ принимает значение 1 только тогда, когда либо a равно 1, а b равно 0, либо a равно 0, а b равно 1. Иначе говоря, $\text{Xor}(a, b) = \text{Or}(\text{And}(a, \text{Not}(b)), \text{And}(\text{Not}(a), b))$. Такое определение реализовано в логической схеме, показанной на рисунке 1.6.

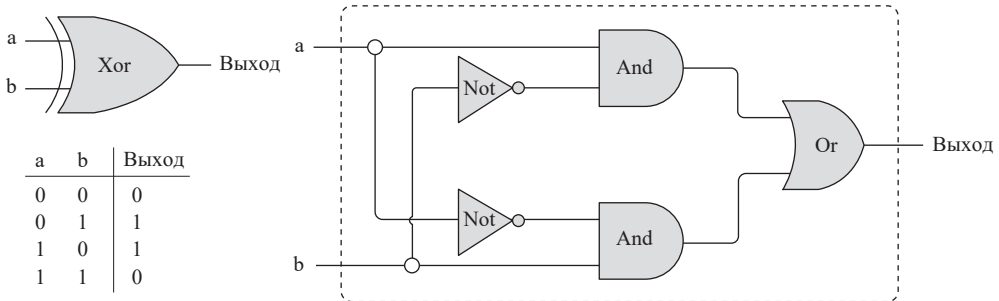


Иллюстрация 1.6. Интерфейс вентиль Хор (слева) и его возможная реализация (справа).

Обратите внимание, что *интерфейс* любого конкретного вентиль уникален: существует только один способ задать его, и обычно это делается с помощью таблицы истинности, булева выражения или словесной спецификации. Однако данный интерфейс можно реализовать множеством различных способов, и некоторые из них будут более элегантными и эффективными, чем другие. Например, реализация Хор, показанная на рисунке 1.6, — это лишь один из возможных вариантов; существуют более эффективные способы реализации Хор, использующие меньше логических вентилей и меньше соединений между ними. Таким образом, с функциональной точки зрения фундаментальное требование логического проектирования заключается в том, чтобы *реализация вентиль была реализацией его заявленного интерфейса, выполненной тем или иным образом*. С точки зрения эффективности общее правило заключается в том, чтобы стараться использовать как можно меньше вентилей, поскольку уменьшение их количества означает падение стоимости и затрат энергии, но повышение скорости вычислений.

Подводя итог, искусство логического проектирования можно описать следующим образом: для данной абстракции вентиль

(называемой также *спецификацией* или *интерфейсом*) надо найти эффективный способ реализации с использованием других, уже реализованных вентиляей.

1.3. Аппаратное конструирование

Теперь мы готовы обсуждать реальное создание вентиляей. Начнем с намеренно упрощенного примера. Предположим, мы открываем мастерскую по производству микросхем в гараже своего дома. Наш первый заказ — изготовить сто вентиляей Хог. Воспользовавшись авансом, мы купили паяльник, моток медной проволоки и три контейнера с надписями «вентили And», «вентили Or» и «вентили Not», каждый из которых содержит множество идентичных копий элементарных логических вентиляей. Каждый вентиль помещен в закрытый пластиковый корпус, и у него открыты только контакты входов, контакт выхода и порт питания. Наша цель — реализовать схему, показанную на иллюстрации 1.6 с помощью данного оборудования.

Для начала возьмем два вентиля And, два вентиля Not и один вентиль Or и установим их на плату в соответствии со схемой. Затем соединим микросхемы друг с другом, прокладывая между ними провода и припаявая концы проводов к соответствующим входным/выходным контактам.

После этого, если мы тщательно воссоздали схему вентиля, у нас останутся три провода с открытыми концами. К каждому из этих концов мы припаиваем по контакту, запечатываем все устройство (за исключением контактов) в пластиковый корпус и маркируем его отметкой «Хог». Этот процесс сборки повторяем множество раз. Под конец складываем все созданные чипы в новый контейнер и приклеиваем к нему ярлык «вентили Хог». Если в будущем нам понадобится сделать какие-то другие чипы, мы сможем воспользоваться этими вентилями Хог как строительными блоками в виде «черного ящика» точно так же, как раньше пользовались вентилями And, Or и Not.

Как вы, вероятно, уже догадались, такой «гаражный» подход к производству микросхем оставляет желать лучшего. Прежде всего, нет

никакой гарантии, что заданная схема является правильной. В таких простых случаях, как Хог, конечно, можно доказать ее правильность, но для многих реально используемых микросхем это невозможно. В результате нам приходится довольствоваться практическим тестированием: собираем микросхему, подключаем ее к источнику питания, активируем и деактивируем входные контакты в различных конфигурациях и надеемся на то, что сигналы на входах и выходе будут соответствовать требуемой спецификации. Если микросхема не справится с этой задачей, нам придется повозиться с ее физической структурой, а это довольно муторное занятие. Кроме того, даже если мы разработаем правильный и эффективный дизайн, многократное повторение процесса сборки микросхемы займет много времени и приведет к ошибкам. Должен же быть лучший способ!

1.3.1. Язык описания аппаратуры

В настоящее время разработчики аппаратуры голыми руками уже ничего не собирают. Вместо этого они проектируют архитектуру чипа с помощью формального средства, называемого *языком описания аппаратуры*, или HDL (*Hardware Description Language*). Разработчик определяет логику чипа, записывая программу на языке HDL, которая затем подвергается строгой проверке. Тесты проводятся виртуально с помощью компьютерного моделирования: специальный программный инструмент, называемый *симулятором аппаратного обеспечения* (*симулятор аппаратуры*), принимает на вход HDL-программу и создает программное представление логики микросхемы. Затем разработчик может дать симулятору команду протестировать виртуальную микросхему на различных наборах входных сигналов. Симулятор вычисляет значения на выходах микросхемы, которые затем сравниваются со значениями, заданными клиентом, заказавшим создание микросхемы.

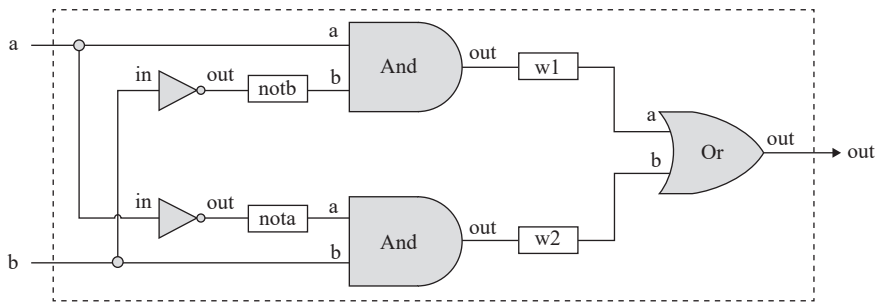
Помимо проверки корректности работы чипа, разработчика аппаратуры обычно интересуют различные параметры, такие как скорость вычислений, энергопотребление и общая стоимость предлагаемой реализации микросхемы. Все эти параметры могут быть смоделированы

и количественно оценены аппаратным симулятором, что помогает разработчику оптимизировать проект до тех пор, пока смоделированный чип не будет соответствовать требуемым уровням стоимости/производительности.

Таким образом, с помощью HDL можно полностью спланировать, отладить и оптимизировать модель микросхемы, не тратя никаких денег на ее физическое производство. Как только клиент одобрит модель чипа, удовлетворяющую его требованиям, оптимизированная версия HDL-программы становится схемой, по которой можно будет отштамповать множество копий физического чипа. Этот последний шаг в процессе проектирования микросхем — от оптимизированной HDL-программы до массового производства — обычно передается на аутсорсинг компаниям, специализирующимся на роботизированном производстве микросхем, основанных на той или иной технологии.

Пример. Построение вентиля Хог. Оставшаяся часть этого раздела представляет собой краткое введение в HDL на примере вентиля Хог; подробную спецификацию HDL можно найти в приложении 2.

Обратим внимание на левую нижнюю часть иллюстрации 1.7. Это HDL-определение микросхемы, состоящее из раздела *заголовка* и разделов *частей*. Заголовок задает *интерфейс* микросхемы, указывая название микросхемы и названия ее входов (IN) и выходов (OUT). В разделе частей (PARTS) перечисляются части микросхемы, из которых состоит ее архитектура. Каждая часть микросхемы задана одним *высказыванием*, в котором указывается название части, затем в скобках следует выражение, в котором показано, как эта часть связана с другими частями проекта. Обратите внимание, что для написания таких высказываний HDL-программист должен иметь доступ к интерфейсам всех базовых частей-чипов: ему нужно знать названия входов и выходов данных частей, а также иметь представление о том, какие операции они производят. Например, программист, написавший программу на иллюстрации 1.7, должен был знать, что входной и выходной контакты вентиля Not называются *in* и *out* и что контакты вентилях And и Or называются *a*, *b* и *out*. (API всех чипов, используемых в программе «От Nand до “Тетриса”», перечислены в приложении 4.)



HDL-программа (Xor.hdl)	Тестовый скрипт (Xor.tst)	Файл вывода (Xor.out)																		
<pre>/* Xor (exclusive or) gate: If a!\b out\1 else out\0. */ CHIP Xor { IN a, b; OUT out; PARTS: Not (in\a, out\nota); Not (in\b, out\notb); And (a\a, b\notb, out\w1); And (a\nota, b\b, out\w2); Or (a\w1, b\w2, out\out); }</pre>	<pre>load Xor.hdl, output-list a, b, out; set a 0, set b 0, eval, output; set a 0, set b 1, eval, output; set a 1, set b 0, eval, output; set a 1, set b 1, eval, output;</pre>	<table><tr><th>a</th><th>b</th><th>out</th></tr><tr><td colspan="3">-----</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	out	-----			0	0	0	0	1	1	1	0	1	1	1	0
a	b	out																		

0	0	0																		
0	1	1																		
1	0	1																		
1	1	0																		

Иллюстрация 1.7. Схема вентиля и HDL-реализация булевой функции Xor (a, b) = Or (And (a , Not (b)), And (Not (a), b)), используемой в качестве примера. Здесь же показаны тестовый скрипт (сценарий тестирования) и файл вывода, сгенерированный в результате тестирования. Подробные описания HDL и языка тестирования приведены в приложениях 2 и 3 соответственно.

Соединения между частями задаются посредством создания и подключения *внутренних контактов* по мере необходимости. Рассмотрим, например, нижнюю часть схемы, где выход вентиля Not подключается ко входу последующего вентиля And. В коде HDL это соединение описывается парой высказываний Not (... , out = nota) и And(a = nota, ...). Первое высказывание создает внутренний контакт (исходящее соединение) с именем nota и передает в него значение вывода out. Второе высказывание передает значение nota на вход a вентиля And. Здесь уместны два комментария. Во-первых, внутренние контакты создаются «автоматически», когда впервые появляются в HDL-программе. Во-вторых, контакты могут неограниченно разветвляться. Например, на иллюстрации 1.7 каждый вход

одновременно подается на два вентиля. На схемах вентиляей множественные соединения задаются посредством изображения разветвлений. В HDL-программах наличие разветвлений определяется кодом.

Язык HDL, который мы используем в программе «От Nand до “Тетриса”», похож на промышленные языки HDL, но гораздо проще. Синтаксис нашего языка HDL в основном не требует пояснений — его можно освоить, просмотрев несколько примеров и при необходимости обратившись к приложению 2.

Тестирование

Строгие стандарты обеспечения качества требуют, чтобы микросхемы тестировались определенным, воспроизводимым и хорошо документированным способом. Исходя из этого симуляторы аппаратного обеспечения обычно включают в себя возможность выполнения *тестовых сценариев* (тестовых скриптов), написанных на языке этих сценариев. Тестовый сценарий, показанный на иллюстрации 1.7, написан на языке, понятном симулятору аппаратного обеспечения «От Nand до “Тетриса”».

Рассмотрим вкратце данный тестовый сценарий. Первые две строки предписывают симулятору загрузить программу `Xor.hdl` и подготовиться к печати значений выбранных переменных. Далее сценарий перечисляет ряд вариантов тестирования. В каждом варианте он приказывает симулятору связать входы микросхемы с выбранными значениями данных (`set`), вычислить значение на выходе (`eval`) и записать результаты тестирования в указанный файл вывода (`output`). В случае простых вентиляей, таких как `Xor`, можно написать исчерпывающий сценарий тестирования, перечисляющий все варианты значений на входах вентиля. В этом случае полученный файл вывода (правая нижняя часть иллюстрации 1.7) служит полной эмпирической проверкой того, что чип работает правильно. Но в более сложных микросхемах, как мы увидим позже, о такой уверенности остается только мечтать.

Читателям, планирующим собрать компьютер Hack, будет приятно узнать, что для всех описанных в книге микросхем в программном пакете «От Nand до “Тетриса”» есть «скелетные» (каркасные)

HDL-программы и тестовые сценарии. В отличие от HDL, который необходимо изучать для того, чтобы составлять спецификации микросхем, наш язык тестирования изучать не нужно. В то же время вы должны уметь читать и понимать прилагаемые тестовые сценарии. Язык сценариев описан в приложении 3, с которым можно ознакомиться по мере необходимости.

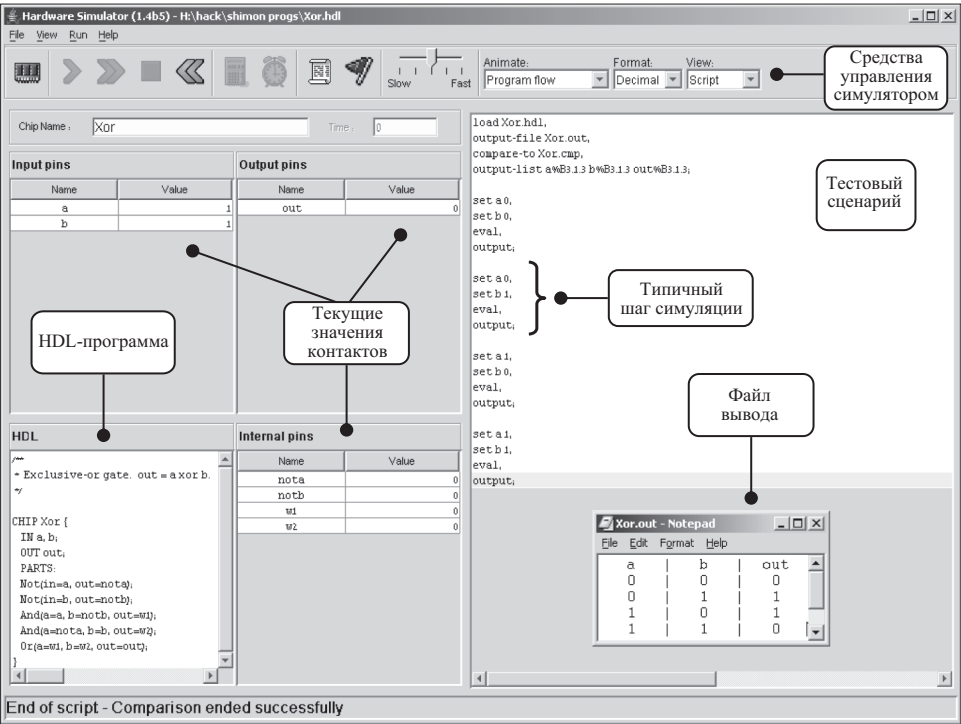


Иллюстрация 1.8. Снимок экрана симуляции чипа Хог в прилагаемом симуляторе аппаратного обеспечения (другие версии этого симулятора могут иметь немного другой графический интерфейс). Здесь отображена стадия сразу же после завершения работы тестового сценария. Значения выводов соответствуют последнему шагу симуляции ($a = b = 1$). На этом снимке не показан файл сравнения, в котором перечислены ожидаемые результаты симуляции, заданные данным тестовым сценарием. Как и тестовый сценарий, файл сравнения обычно предоставляется клиентом, желающим получить микросхему. В данном конкретном примере файл вывода, созданный в результате симуляции (правая нижняя часть иллюстрации), идентичен предоставленному файлу сравнения.

1.3.2. Симуляция аппаратного обеспечения

Написание и отладка программ на HDL похожи на обычную разработку программного обеспечения. Основное отличие заключается в том, что вместо написания кода на высокоуровневом языке мы пишем его на HDL, а вместо компиляции и запуска кода используем симулятор аппаратуры для его тестирования. Симулятор аппаратного обеспечения (аппаратуры) — это компьютерная программа, которая умеет разбирать и интерпретировать HDL-код, превращать его в набор команд и проводить тестирование в соответствии с предоставленными тестовыми сценариями. На рынке существует множество коммерческих симуляторов аппаратуры. Программный пакет «От Nand до “Тетриса”» включает в себя простой симулятор аппаратуры, предоставляющий все необходимые инструменты для создания, тестирования и интеграции всех описанных в книге микросхем, вплоть до создания универсального компьютера. На иллюстрации 1.8 показан типичный сеанс моделирования микросхемы.

1.4. Спецификация

Перейдем теперь к определению спецификаций набора логических вентилях, которые понадобятся для построения микросхем нашей компьютерной системы. Это обычные вентили, каждый из которых предназначен для выполнения обычной булевой операции. Для каждого вентиля мы сосредоточимся на его интерфейсе (*что* должен делать этот вентиль), отложив детали реализации (*как* сделать так, чтобы он работал) до последующего раздела.

1.4.1. Nand (И-НЕ)

Отправной точкой нашей компьютерной архитектуры будет вентиль Nand (И-НЕ), на основе которого будут построены все остальные вентили и микросхемы. Вентиль Nand реализует следующую булеву функцию.

a	b	$\text{Nand}(a, b)$
0	0	1
0	1	1
1	0	1
1	1	0

Или, если это выразить в стиле API:

```
Chip name: Nand
Input: a, b
Output: out
Function: if ((a==1) and (b==1)) then out = 0, else out = 1
```

На протяжении всей книги микросхемы задаются с помощью показанного выше стиля API. Для каждой микросхемы в API указываются название микросхемы (Chip Name), названия ее входов (Input) и выходов (Output), предполагаемая функция (Function) или операция микросхемы, а также необязательные комментарии.

1.4.2. Базовые логические вентили

Перечисленные далее логические вентили обычно называют базовыми, поскольку они используются при создании более сложных микросхем. Вентили Not, And, Or и Xor реализуют классические логические операторы, а мультиплексор и демультиплексор обеспечивают средства управления потоками информации.

Not (НЕ): этот вентиль, известный также под названием «инвертор», на выходе выводит значение, противоположное значению на входе. Вот его определение в стиле API:

```
Chip name: Not
Input: in
Output: out
Function: if (in==0) then out = 1, else out = 0
```

And (И): возвращает 1, если значения на обоих входах равны 1, в противном случае возвращает 0:

Chip name: And

Input: a, b

Output: out

Function: if ((a==1) and (b==1)) then out = 1, else out = 0

Or (ИЛИ): возвращает 1, когда значение по меньшей мере на одном входе равно 1, в противном случае возвращает 0:

Chip name: Or

Input: a, b

Output: out

Function: if ((a==0) and (b==0)) then out = 0, else out = 1

Xor (исключающее ИЛИ): возвращает 1, когда значение строго на одном из входов равно 1, в противном случае возвращает 0:

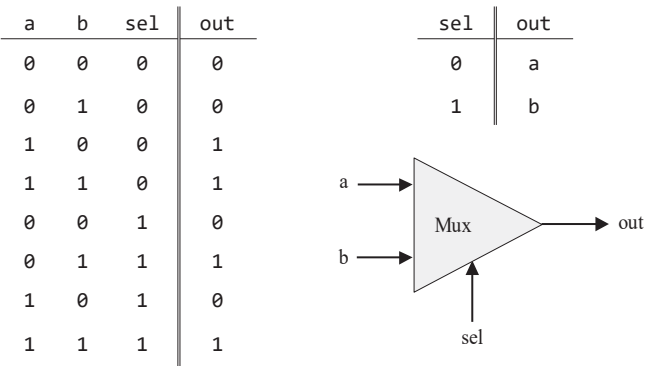
Chip name: Xor

Input: a, b

Output: out

Function: if (a!=b) then out = 1, else out = 0

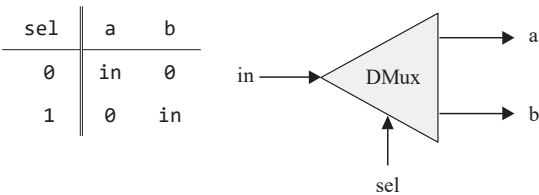
Мультиплексор: мультиплексор — это вентиль с тремя входами (см. иллюстрацию 1.9). Два входных бита, названных a и b, интерпретируются как *биты данных*, а третий входной бит, названный sel, интерпретируется как бит выбора. Вход sel служит для выбора и направления на выход значения либо a, либо b. Таким образом, разумным названием для этого устройства могло бы быть «селектор» («выбира-тель»). Название «мультиплексор» было позаимствовано из систем связи, где расширенные версии данного устройства используются для сериализации (мультиплексирования) нескольких входных сигналов по одному каналу связи.



Chip name: Mux
Input: a, b, sel
Output: out
Function: if (sel\\0) then out \ a, else out \ b

Иллюстрация 1.9. Мультиплексор. Таблица справа сверху — сокращенная версия таблицы истинности.

Демультимплексор: демультимплексор выполняет функцию, противоположную функции мультиплексора, он принимает одно входное значение и направляет его на один из двух возможных выходов в соответствии с селекторным битом, который выбирает выход назначения. На другом выходе устанавливается значение 0. На иллюстрации 1.10 показан API.



Chip name: DMux
Input: in, sel
Output: a, b
Function: if (sel\\0) then {a, b} \ {in, 0},
 else {a, b} \ {0, in}

Иллюстрация 1.10. Демультимплексор.

1.4.3. Многоразрядные версии основных вентилях

Компьютерное оборудование часто предназначено для обработки многоразрядных значений, например, для вычисления побитовой функции *И* для чисел на двух 16-битных входах. В этом разделе описывается несколько 16-битных логических вентилях, которые понадобятся для построения нашей компьютерной платформы. Попутно отметим, что логическая архитектура этих *n*-битных вентилях одинакова независимо от значения *n* (например, 16, 32 или 64 бита). HDL-программы обращаются с многоразрядными значениями как с однокбитовыми, за исключением того, что значения могут быть проиндексированы для доступа к отдельным битам. Например, если *in* и *out* представляют 16-битные значения, то выражение `out[3]=in[5]` присваивает третьему биту *out* значение пятого бита *in*. Биты индексируются справа налево, крайний правый бит — это нулевой бит, а крайний левый — 15-й бит (для 16-битной архитектуры).

Многоразрядный Not (НЕ): *n*-битный вентиль **Not** применяет булеву операцию **Not** к каждому из битов на своем *n*-битном входе:

```
Chip name: Not16
Input: in[16]
Output: out[16]
Function: for i = 0..15 out[i] = Not(in[i])
```

Многоразрядный And (И): *n*-битный вентиль **And** применяет булеву операцию **And** к каждой паре битов на своих двух *n*-битных входах:

```
Chip name: And16
Input: a[16], b[16]
Output: out[16]
Function: for i = 0..15 out[i] = And(a[i], b[i])
```

Многоразрядный Or (ИЛИ): n -битный вентиль **Or** применяет булеву операцию **Or** к каждой паре битов на своих двух n -битных входах:

Chip name: Or16

Input: a[16], b[16]

Output: out[16]

Function: for i = 0..15 out[i] = Or(a[i], b[i])

Многоразрядный мультиплексор: n -битный мультиплексор действует так же, как и базовый мультиплексор, за исключением того, что его входы и выходы принимают n -битные значения:

Chip name: Mux16

Input: a[16], b[16], sel

Output: out[16]

Function: if (sel==0) then for i = 0..15 out[i] = a[i],
else for i = 0..15 out[i] = b[i]

1.4.4. Версии базовых вентилях со множеством входов и выходов

Логические вентили, имеющие один или два входа, допускают естественное обобщение и для большего количества входов (более двух). В этом разделе описывается набор многовходовых вентилях, которые в дальнейшем будут использоваться в различных микросхемах нашей компьютерной архитектуры.

Многовходовый Or (ИЛИ): m -входовый вентиль **Or** выдает 1, если по меньшей мере на один из его m входов подается значение 1, в противном случае выдает 0. Нам понадобится восьмивходовый (8-way) вариант этого вентиля:

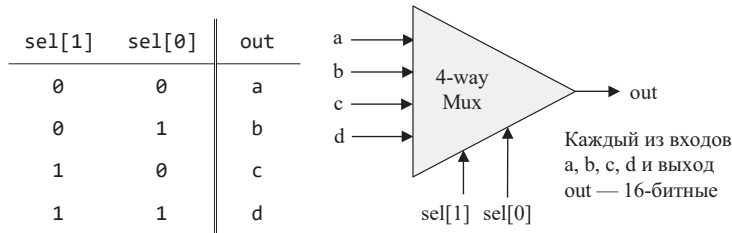
Chip name: Or8Way

Input: in[8]

Output: out

Function: out = Or(in[0], in[1],..., in[7])

Многовходовый/многоразрядный мультиплексор: m -входовый мультиплексор выбирает один из m n -битных входов и выводит его значение на n -битный выход. Выбор задается набором из k битов выбора, где $k = \log_2 m$. Вот API 4-входового мультиплексора.



Для нашей компьютерной платформы потребуются два варианта этого чипа: 4-входовый 16-битный мультиплексор и 8-входовый 16-битный мультиплексор:

Chip name: Mux4Way16

Input: a[16], b[16], c[16], d[16], sel[2]

Output: out[16]

Function: if (sel==00,01,10, or 11) then out = a, b, c, or d

Comment: The assignment is a 16-bit operation.

For example, "out = a" means "for i = 0..15
out[i] = a[i]".

(Это 16-битная операция. Например, "out = a" означает
"for i = 0..15 out[i] = a[i]".)

Chip name: Mux8Way16

Input: a[16], b[16], c[16], d[16], e[16], f[16],
g[16], h[16], sel[3]

Output: out[16]

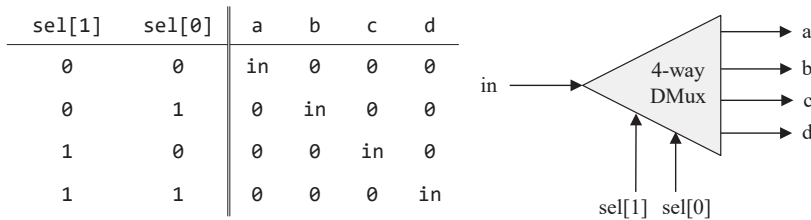
Function: if (sel==000,001,010, ..., or 111) then out = a,
b, c, d, ..., or h

Comment: The assignment is a 16-bit operation.

For example, "out = a" means "for i = 0..15
out[i] = a[i]".

(Это 16-битная операция. Например, "out = a" означает
"for i = 0..15 out[i] = a[i]".)

Многовыходовой/многоразрядный демультиплексор: m -входовый n -битный демультиплексор перенаправляет n -битное значение входа на один из m n -битных выходов. Для других выходов устанавливается значение 0. Выбор задается набором из k битов выбора, где $k = \log_2 m$. Вот API 4-входового демультиплексора.



Для нашей компьютерной платформы потребуются два варианта этого чипа: 4-выходовой 1-битный демультиплексор и 8-выходовой 1-битный демультиплексор:

Chip name: DMux4Way

Input: in, sel[2]

Output: a, b, c, d

```
Function:  if (sel==00) then {a, b, c, d} = {1,0,0,0},
           else if (sel==01) then {a, b, c, d} = {0,1,0,0},
           else if (sel==10) then {a, b, c, d} = {0,0,1,0},
           else if (sel==11) then {a, b, c, d} = {0,0,0,1}
```

Chip name: Dmux8Way

Input: in, sel[3]

Output: a, b, c, d, e, f, g, h

```
Function:  if (sel==000) then {a, b, c,..., h} =
           {1,0,0,0,0,0,0,0},
           else if (sel==001) then {a, b, c,..., h} =
           {0,1,0,0,0,0,0,0},
           else if (sel==010) then {a, b, c,..., h} =
           {0,0,1,0,0,0,0,0},
           ...
           else if (sel==111) then {a, b, c,..., h} =
           {0,0,0,0,0,0,0,1}
```

1.5. Реализация

В предыдущем разделе были описаны спецификации (или интерфейсы) семейства базовых логических вентилях. Описав «что», мы переходим к обсуждению «как». В частности, сосредоточимся на двух общих подходах к реализации логических вентилях: *поведенческой симуляции* и *аппаратной реализации*. Оба подхода играют важную роль во всех наших проектах по созданию аппаратного обеспечения.

1.5.1. Моделирование поведения

Представленные до сих пор описания микросхем были строго абстрактными. Было бы неплохо поэкспериментировать с этими абстракциями на практике, прежде чем приступить к их созданию на языке HDL. Но как можно это сделать?

Если мы хотим только протестировать работу и взаимодействие микросхем, то необязательно создавать их на языке HDL. Вместо этого можно выбрать гораздо более простую реализацию на основе обычного программирования. Например, воспользоваться каким-нибудь объектно-ориентированным языком для создания набора классов, каждый из которых будет реализовывать типовой чип. Можно написать конструкторы классов для создания экземпляров микросхем и методы *eval* для оценки их логики, а затем заставить классы взаимодействовать друг с другом так, чтобы микросхемы более высокого уровня определялись в терминах микросхем более низкого уровня. После этого можно добавить красивый графический интерфейс, позволяющий подавать различные значения на входы микросхем, оценивать их логику и наблюдать за выходами. Такой программный метод, называемый *симуляцией (моделированием) поведения*, имеет большой смысл. Он позволяет поэкспериментировать с интерфейсами микросхем до начала трудоемкого процесса их создания на языке HDL.

Симулятор аппаратуры «От Nand до “Тетриса”» предоставляет именно такую услугу. Помимо симуляции поведения HDL-программ, что является его основной задачей, он имеет встроенные программные реализации всех микросхем, создаваемых в аппаратных проектах

«От Nand до “Тетриса”». Встроенная версия каждого чипа реализована в виде исполняемого программного модуля, вызываемого каркасной HDL-программой, обеспечивающей интерфейс чипа. Например, вот HDL-программа, реализующая встроенную версию микросхемы Xor:

```
/* Xor (exclusive or) gate:
   If a!=b out=1 else out=0. */
CHIP Xor {
    IN a, b;
    OUT out;
    BUILTIN Xor;
}
```

Сравните это с программой на языке HDL, приведенной на иллюстрации 1.7. Во-первых, обратите внимание, что обычные и встроенные микросхемы имеют абсолютно одинаковый интерфейс. Таким образом, они обеспечивают абсолютно одинаковую функциональность. Однако во встроенной реализации секция PARTS заменена единственным высказыванием BUILTIN Xor. Это высказывание сообщает симулятору, что микросхема реализована с помощью Xor.class. Этот файл класса, как и все файлы классов Java, реализующие встроенные микросхемы, находится в папке nand2tetris/tools/builtIn.

Вскользь отметим, что реализовать логические вентили с помощью высокоуровневых языков программирования несложно и это еще одно преимущество моделирования поведения: оно недорогое и быстрое. В какой-то момент, конечно, инженеры по аппаратному обеспечению должны заняться настоящим делом — реализацией микросхем не в виде программных артефактов, а в виде HDL-программ, результатом работы которых будут физические кремниевые чипы. И мы далее займемся этим.

1.5.2. Аппаратная реализация

В этом разделе приведены рекомендации по реализации описанных в данной главе пятнадцати логических вентилях. Как правило, наши рекомендации по реализации в этой книге намеренно кратки. Мы

даем достаточно информации для начала работы, оставляя вам удовольствие самостоятельно разобраться с остальными реализациями вентиляей.

Nand: поскольку мы решили положить в основу нашего устройства элементарные вентили Nand, будем рассматриваем Nand как примитивный вентиль, функциональность которого задана извне. Поставляемый аппаратный симулятор имеет встроенную реализацию Nand, поэтому необходимости его реализовывать нет.

Not: может быть реализован с помощью одного вентиля Nand. Совет: посмотрите на таблицу истинности Nand и спросите себя, как можно расположить входы Nand так, чтобы при одном входном сигнале 0 вентиль Nand выводил 1, а при одном входном сигнале 1 выводил 0.

And: можно реализовать с помощью двух рассмотренных выше вентиляей.

Or / Xor: булеву функцию **Or** можно задать с помощью булевых функций **And** и **Not**. Булеву функцию **Xor** можно задать с помощью **And**, **Not** и **Or**.

Мультиплексор / демультиплексор: можно реализовать с использованием ранее построенных вентиляей.

Многоразрядные вентили Not / And / Or: если вы уже построили базовые версии данных вентиляей, реализация их n -арных версий — это вопрос организации массивов из n -базовых вентиляей и отдельной работы каждого вентиля на своих однобитовых входах. Получившийся HDL-код будет несколько скучным и повторяющимся (с использованием копирования и вставки), но он важен тем, что позже эти многоразрядные вентили будут использоваться при построении более сложных микросхем.

Многоразрядный мультиплексор: реализация n -арного мультиплексора сводится к подаче одного и того же бита выбора на каждый

из n -двоичных мультиплексоров. Опять же, скучная конструкторская задача, в результате которой получается очень полезная микросхема.

Многовходовые вентили. Совет по реализации: подумайте о разветвлениях.

1.5.3. Встроенные микросхемы

Как мы уже отмечали при обсуждении моделирования поведения, наш симулятор аппаратуры имеет программные встроенные реализации большинства описанных в книге микросхем. Наиболее распространенная микросхема «От Nand до “Тетриса”» — это, конечно же, Nand: всякий раз, когда вы в своей HDL-программе используете в качестве части какой-то микросхемы Nand, аппаратный симулятор вызывает встроенную реализацию `tools/builtIn/Nand.hdl`. Это частный случай более общей стратегии вызова микросхем: всегда, когда аппаратный симулятор встречает в HDL-программе некую микросхему-часть, скажем, *Xxx*, он ищет файл *Xxx.hdl* в текущей папке; если файл найден, симулятор оценивает лежащий в его основе HDL-код. Если файл не найден, симулятор ищет его в папке `tools/builtIn`. Если файл найден, симулятор выполняет встроенную реализацию микросхемы; в противном случае он выдает сообщение об ошибке и завершает симуляцию.

Такая стратегия бывает очень удобной. Предположим, например, что вы начали реализовывать программу `Mux.hdl`, но по какой-то причине не завершили ее. Это может стать досадной неудачей, потому что теоретически вы не сможете продолжить создание микросхем, где в качестве частей используются чипы Mux. К счастью (как на самом деле и было задумано), здесь на помощь приходят встроенные микросхемы. Вам нужно будет только переименовать свою частичную реализацию, например, в `Mux1.hdl`. Каждый раз при моделировании функциональности микросхемы с составной частью Mux симулятор аппаратуры постарается вызвать из текущей папки файл `Mux.hdl`, но не найдет его. Так симулятор вызовет вместо него встроенную версию Mux, и в действие вступит моделирование поведения. Как раз то, что и требовалось! В дальнейшем можно будет вернуться к `Mux1.hdl`

и продолжить работу по его реализации. В таком случае вы просто вернете файлу изначальное название `Mux.hdl` и завершите работу.

1.6. Проект

В этом разделе описаны инструменты и ресурсы, необходимые для выполнения проекта 1, а также приведены рекомендуемые шаги и советы по выполнению.

Задача: реализовать все логические вентили, представленные в данной главе. Единственные строительные блоки, которые можно использовать, — это примитивные вентили Nand и составные вентили, которые вы будете постепенно создавать на их основе.

Ресурсы: мы предполагаем, что вы уже скачали zip-файл «От Nand до “Тетриса”» (Nand to Tetris), содержащий набор программного обеспечения книги, и распаковали его в папку с именем `nand2tetris`. Если это так, то папка `nand2tetris/tools` на вашем компьютере содержит аппаратный симулятор, о котором говорилось в данной главе. Эта программа и обычный текстовый редактор — единственные инструменты, необходимые для выполнения проекта 1, а также всех остальных описанных в книге аппаратных проектов.

Пятнадцать микросхем, упомянутых в данной главе, за исключением Nand, нужно реализовать на языке HDL, описанном в приложении 2. Для каждой микросхемы `Xxx` мы предоставляем каркасную программу `Xxx.hdl` (иногда называемую «файлом-заглушкой») с недостающей частью реализации. Кроме того, для каждой микросхемы предоставлен сценарий `Xxx.tst`, сообщающий аппаратному симулятору, как ее тестировать, а также файл сравнения `Xxx.cmp`, где перечислены правильные выходные данные, которые должен выдать тест. Все эти файлы доступны в папке `nand2tetris/projects/01` у вас на компьютере.

Ваша задача — заполнить и протестировать все файлы `Xxx.hdl` в данной папке. Эти файлы можно открывать и редактировать в любом обычном текстовом редакторе.

Контракт (условие задачи): при загрузке в симулятор аппаратуры ваша конструкция микросхемы (модифицированная программа `.hdl`), протестированная на прилагаемом файле `.tst`, должна выдать такую же схему выходов, как и в прилагаемом файле `.scr`. Если фактические выходные значения, сгенерированные симулятором, расходятся с требуемыми, симулятор остановит симуляцию и выдаст сообщение об ошибке.

Шаги. Мы советуем выполнять задание в следующем порядке.

0. Необходимый для этого проекта *симулятор аппаратуры* содержится в папке `nand2tetris/tools`.
1. По мере необходимости сверяйтесь с приложением 2 (описание языка HDL).
2. По мере необходимости сверяйтесь с учебным руководством симулятора аппаратуры (Hardware Simulator Tutorial, доступным на сайте www.nand2tetris.org).
3. Постройте все чипы, перечисленные в папке `nand2tetris/projects/01`, и проведите их симуляцию.

Общие советы по реализации

Мы используем слова «*вентиль*», «*чип*» и «*микросхема*» как взаимозаменяемые.

- Каждый вентиль можно реализовать более чем одним способом. Чем проще реализация, тем лучше. В качестве общего правила: старайтесь использовать как можно меньше чипов-частей.
- Хотя каждый чип можно реализовать непосредственно с помощью одних лишь вентилях Nand, мы советуем всегда использовать уже реализованные составные вентили. Смотрите совет выше.
- Нет необходимости создавать «вспомогательные чипы» собственной разработки. Ваши HDL-программы должны использовать только чипы, упомянутые в этой главе.

— Реализуйте чипы в том порядке, в котором они появляются в данной главе. Если по какой-то причине вы не завершите HDL-реализацию какого-либо чипа, то все равно сможете использовать его как часть микросхемы в других HDL-программах. Просто переименуйте файл чипа или удалите его из папки, заставив симулятор использовать вместо него встроенную версию.

Веб-версия проекта 1 доступна на сайте www.nand2tetris.org.

1.7. Перспектива

В этой главе был определен набор основных логических вентилей, широко используемых в компьютерных архитектурах. В главах 2 и 3 с помощью данных вентилей мы построим микросхемы для обработки и хранения данных соответственно. Эти микросхемы, в свою очередь, в дальнейшем будут использованы для построения центрального процессора и устройств памяти нашего компьютера.

Хотя мы выбрали в качестве основного строительного блока вентиль Nand, в качестве возможных базовых блоков можно использовать и другие логические вентили. Например, можно построить полную компьютерную платформу с использованием только вентилей Nor или, в качестве альтернативы, комбинации вентилей And, Or и Not. Эти конструктивные подходы к проектированию логики теоретически эквивалентны, подобно тому как одна и та же геометрия может основываться на альтернативных наборах заданных аксиом. В принципе, если инженеры-электрики или физики смогут придумать эффективные и недорогие реализации логических вентилей на основе любой технологии, которую они сочтут необходимой, мы с удовольствием воспользуемся ими в качестве примитивных строительных блоков. Реальность же такова, что большинство компьютеров построено на основе либо вентилей Nand, либо вентилей Nor.

На протяжении всей главы мы не обращали внимания на соображения эффективности и стоимости, такие как энергопотребление или количество пересечений проводов, подразумеваемых

нашими HDL-программами. Но такие соображения критически важны на практике, и большая часть сферы компьютерных технологий посвящена как раз их оптимизации. Еще один вопрос, который мы не рассматривали, — это физические аспекты реализации, например, то, как примитивные логические вентили могут создаваться на основе транзисторов в кристалле кремния или на основе других технологий, позволяющих обрабатывать сигналы. Конечно, существует несколько вариантов таких реализаций, каждый из которых имеет свои характеристики (скорость, энергопотребление, стоимость производства и т. д.). Любое нетривиальное освещение этих вопросов требует обращения к областям за пределами компьютерной науки, таким как электротехника и физика твердого тела.

В следующей главе рассказывается о том, как биты используются для представления двоичных чисел и как логические вентили используются для реализации арифметических операций. Эти возможности основаны на принципах работы элементарных логических вентилей, построенных в данной главе.

2. Булева арифметика

Счет — религия этого поколения, его надежда и спасение.

— *Гертруда Стайн (1874–1946)*

В этой главе мы создадим семейство микросхем, предназначенных для представления чисел и выполнения арифметических операций. Нашей отправной точкой будет набор логических вентилях, построенных в главе 1, а конечной точкой — полностью функциональное *арифметико-логическое устройство* (АЛУ). Впоследствии АЛУ станет вычислительным центром центрального процессора, или *центрального процессорного устройства* (ЦПУ) — микросхемы, выполняющей все команды, обрабатываемые компьютером. Таким образом, создание АЛУ — это важная веха в нашем путешествии «От Nand до “Тетриса”».

Как обычно, мы подходим к задаче постепенно, начиная с вступительного раздела, в котором описывается, как двоичные коды и булева арифметика используются соответственно для представления и сложения целых чисел со знаком. В разделе «Спецификация» представлена последовательность *сумматоров*, предназначенных для сложения двух битов, трех битов и пар n -битных двоичных чисел. Так подготавливается база для спецификации АЛУ, основанного на удивительно простой логической схеме. Разделы «Реализация» и «Проект» содержат советы и рекомендации по созданию микросхем сумматоров и АЛУ с использованием языка HDL и прилагаемого симулятора аппаратуры.

2.1. Арифметические операции

Компьютерные системы общего назначения должны выполнять как минимум следующие арифметические операции над целыми числами со знаком (положительными и отрицательными):

- сложение;
- изменение знака;
- вычитание;
- сравнение;
- умножение;
- деление.

Мы начнем с разработки логики вентилях, выполняющих сложение и изменение знака. Позже будет показано, как на основе этих двух строительных блоков можно реализовать другие арифметические операции.

В математике, как и в информатике, *сложение* — это простая операция, имеющая глубокий смысл. Примечательно, что к сложению двоичных чисел можно свести все выполняемые цифровыми компьютерами функции, а не только арифметические операции. Поэтому конструктивное понимание принципов двоичного сложения служит ключом к пониманию многих фундаментальных операций, выполняемых аппаратным обеспечением компьютера.

2.2. Двоичные числа

Когда говорят, что определенный код, скажем, 6083, представляет число в десятичной системе счисления, то, согласно установленным правилам, мы интерпретируем это следующим образом:

$$(6083)_{10} = 6 \cdot 10^3 + 0 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0 = 6083$$

Каждая цифра в десятичном коде соответствует значению, зависящему от положения цифры в коде, соответствующего степени числа 10, то есть оно записано *по основанию 10*. Предположим, нам сказали, что код 10011 представляет число, записанное *по основанию 2*, то есть это число в *двоичном* представлении. Чтобы вычислить значение этого числа, мы выполняем точно такую же процедуру преобразования, как и выше, но используем основание 2 вместо основания 10:

$$(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Внутри компьютеров все представлено в виде двоичных кодов. Например, когда мы в ответ на предложение «Приведите пример простого числа» нажимаем на клавиатуре клавиши 1, 9 и Enter, в памяти компьютера сохраняется двоичный код 10011. Когда мы просим компьютер вывести это значение на экран, происходит следующее. Сначала операционная система компьютера вычисляет десятичное значение 10011, которое оказывается равным 19. После преобразования этого целочисленного значения в два символа 1 и 9, ОС просматривает текущий шрифт и получает два растровых изображения, используемых для отображения данных символов на экране. Затем ОС заставляет драйвер экрана включать и выключать соответствующие пиксели, и через какую-то долю секунды (весь процесс занимает очень мало времени) мы видим, как на экране появляется изображение «19».

В главе 12 мы разработаем операционную систему, которая будет выполнять такие операции по преобразованию и отображению, а также множество других низкоуровневых операций. Пока достаточно заметить, что десятичное представление чисел — это своего рода поправка людям, объясняемая тем фактом, что в какой-то неопределенный момент древней истории они решили отображать количество предметов с помощью десяти пальцев и эта привычка прижилась. С математической точки зрения число десять совершенно неинтересно, а для компьютеров оно и вовсе неудобно. Компьютеры работают со всем в двоичной системе счисления, нисколько не заботясь о десятичной. Тем не менее, поскольку люди настаивают на том, чтобы работать с числами в десятичном представлении, компьютерам приходится

проделывать много дополнительной работы и, прежде чем отобразить для человека результат вычислений на экране, выполнять преобразования из двоичной системы в десятичную и наоборот. В остальное время компьютеры придерживаются двоичной системы счисления.

Фиксированный размер машинного слова. Количество целых чисел, конечно, не ограничено: для любого заданного числа x существуют целые числа, которые меньше x , и целые числа, которые больше x . Тем не менее компьютеры — это машины с ограниченными параметрами, и они для представления чисел используют фиксированный размер машинного слова. «Машинное слово» — распространенный аппаратный термин для обозначения количества битов, используемого для представления базового блока информации, в данном случае некоего целого числа. Обычно для целых чисел используются 8-, 16-, 32- или 64-битовые регистры*. Фиксированный размер слова подразумевает, что существует ограничение на количество значений, которые могут передавать данные регистры.

Предположим, что для представления целых чисел мы используем 8-битные регистры. Всего в таком случае мы можем закодировать $2^8 = 256$ различных чисел. Если мы хотим работать только с неотрицательными числами, то можем присвоить код 00000000 числу 0, код 00000001 числу 1, 00000010 — числу 2, 00000011 — числу 3 и т. д., вплоть до кода 11111111, соответствующего числу 255. В общем случае, используя n бит, мы можем представить все неотрицательные числа в диапазоне от 0 до $2^n - 1$.

Но как же с помощью двоичного кода передавать отрицательные числа? Позже в этой главе мы покажем способ, решающий данную задачу наиболее элегантным и эффективным способом.

* Что соответствует типичным высокоуровневым типам данных *byte*, *short*, *int* и *long*. Например, будучи сведенными до уровня машинных команд, переменные типа *short* могут обрабатываться 16-битными регистрами. Поскольку производить арифметические операции над 16-битными данными в четыре раза быстрее, чем над 64-битными, программистам рекомендуется всегда использовать наиболее компактный тип данных, удовлетворяющий требованиям приложения. — Здесь и далее прим. авт., если не указано иное.

А как насчет чисел, которые больше или меньше максимального и минимального значений, определенных фиксированным размером регистра? Каждый язык высокого уровня имеет абстракции для работы с числами, которые могут быть настолько велики или настолько малы, насколько это практически необходимо. Такие абстракции обычно реализуются посредством объединения некоторого количества n -битных регистров, необходимого для представления нужного числа. Поскольку выполнение арифметических и логических операций над многословными числами — дело медленное, рекомендуется использовать такую практику только тогда, когда приложение действительно требует обработки чрезвычайно больших или чрезвычайно малых чисел.

2.3. Двоичное сложение

Сложение пары двоичных чисел можно осуществлять побитово справа налево, используя алгоритм сложения десятичных чисел, который изучают в школе. Сначала мы складываем два крайних правых бита, которые также называются *наименее значимыми* (*самыми младшими*) битами двух двоичных чисел. Затем добавляем полученный бит переноса к сумме следующей пары битов. Этот процесс продолжается до тех пор, пока не будут сложены два левых *наиболее значимых* (*самых старших*) бита. Вот пример такого алгоритма в действии (в данном случае фиксированный размер слова равен 4 битам):

0	0	0	1	(Перенос)	1	1	1	1	
	1	0	0	x		1	0	1	1
+	0	1	0	y	+	0	1	1	1
0	1	1	1	x + y	1	0	0	1	0
Без переполнения					Переполнение				

Если при побитовом сложении в самом старшем бите получается перенос, равный 1, возникает ситуация так называемого *переполнения*.

Как поступать с переполнением — зависит от того, кто и с какой целью принимает решение. Наш вариант — игнорировать его. В принципе, мы довольствуемся тем, что результат сложения любых двух n -битных чисел будет правильным до n бит. Вскользь отметим, что игнорирование вполне допустимо, если только об этом заявлено ясно и открыто.

2.4. Двоичные числа со знаком

В n -битной двоичной системе можно закодировать 2^n различных чисел. Если нужно представить в двоичном коде знаковые (положительные и отрицательные) числа, то естественным решением будет разделить доступное кодовое пространство на два подмножества: одно для представления неотрицательных чисел, а другое для представления отрицательных чисел. В идеале нужно выбирать такую схему кодирования, чтобы введение знаковых чисел как можно меньше усложняло аппаратную реализацию арифметических операций.

С годами были разработаны несколько схем представления знаковых чисел в двоичной системе. Решение, используемое сегодня почти во всех компьютерах, называется методом *дополнения до двух*, или просто *дополнительным кодом*. Согласно этому методу в двоичной системе с размером слова в n бит отрицательное число x представляется двоичным кодом, соответствующим числу $2^n - x$. Например, в 4-битной двоичной системе число -7 будет представлено двоичным кодом, соответствующим числу $2^4 - 7 = 9$, то есть в виде 1001 . Вспомнив, что $+7$ представлено в виде 0111 , в результате сложения этих чисел получаем $1001 + 0111 = 0000$ (с игнорированием бита переполнения). На иллюстрации 2.1 перечислены все знаковые числа 4-битной системы с использованием метода дополнительного кода.

Проанализировав иллюстрацию 2.1, можно прийти к выводу, что n -битная двоичная система с дополнительными кодами обладает следующими привлекательными свойствами:

- система кодирует 2^n знаковых чисел в диапазоне от $-(2^{n-1})$ до 2^{n-1} ;
- код любого неотрицательного числа начинается с 0;

- код любого отрицательного числа начинается с 1;
- чтобы получить двоичный код $-x$ из двоичного кода x , нужно оставить все наименее значимые биты с 0 и первый наименее значимый бит с 1 нетронутыми, а все остальные инвертировать (вместо 0 поставить 1 и наоборот). В качестве альтернативы можно инвертировать все биты x и добавить к результату 1.

0000:	0	
0001:	1	
0010:	2	
0011:	3	
0100:	4	
0101:	5	
0110:	6	
0111:	7	
1000:	-8	(16 - 8)
1001:	-7	(16 - 7)
1010:	-6	(16 - 6)
1011:	-5	(16 - 5)
1100:	-4	(16 - 4)
1101:	-3	(16 - 3)
1110:	-2	(16 - 2)
1111:	-1	(16 - 1)

Иллюстрация 2.1. Представление знаковых чисел в 4-битной двоичной системе методом дополнения до двух.

Особенно в методе дополнительного кода привлекает то, что вычитание рассматривается как частный случай сложения. Для иллюстрации рассмотрим пример $5 - 7$. Это выражение эквивалентно выражению $5 + (-7)$, и, руководствуясь иллюстрацией, выполним сложение $0101 + 1001$. В результате получается 1110 , и это действительно двоичный код числа -2 . Приведем еще один пример. Чтобы вычислить $(-2) + (-3)$, сложим $1110 + 1101$ и получим результат 11011 . Пройгнорировав бит переполнения, получаем 1011 , что и есть двоичный код числа -5 .

Видно, что метод дополнительного кода (дополнения до двух) позволяет складывать и вычитать знаковые числа с использованием одних лишь аппаратных средств, необходимых для сложения неотрицательных чисел.

Как будет показано далее в книге, каждую арифметическую операцию, от умножения до деления и извлечения квадратного корня, можно свести к двоичному сложению. Итак, с одной стороны, мы видим, что огромный спектр возможностей компьютера опирается на двоичное сложение, а с другой стороны — что метод дополнительного кода избавляет от необходимости в специальном оборудовании для сложения и вычитания знаковых чисел. На основании этих наблюдений мы вынуждены признать, что метод дополнительного кода — одно из самых замечательных достижений прикладной информатики, редко удостоивающееся заслуженного внимания.

2.5. Спецификация

Перейдем теперь к определению иерархии микросхем, начиная с простых сумматоров и заканчивая арифметико-логическим устройством (АЛУ). Как принято в нашей книге, сначала сосредоточимся на абстрактном (для чего предназначены микросхемы), отложив детали реализации (как они это делают) до следующего раздела. Не можем удержаться, чтобы с удовольствием не повторить, что благодаря методу дополнительного кода нам не нужно изобретать ничего особенного для обработки знаковых чисел. Все арифметические микросхемы, которые мы представим, одинаково хорошо работают с неотрицательными, отрицательными и смешанными знаковыми числами.

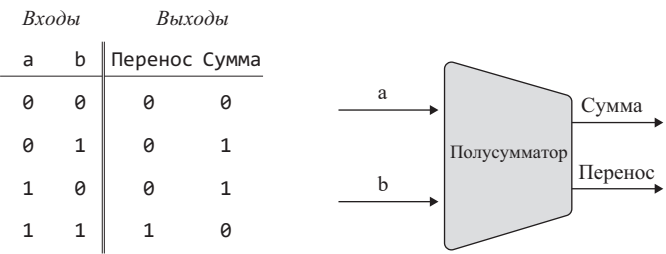
2.5.1. Сумматоры

Сосредоточимся на следующей иерархии сумматоров.

- *Полусумматор*: предназначен для сложения двух битов.
- *Полный сумматор*: предназначен для сложения трех битов.
- *Сумматор*: предназначен для сложения двух n -битных чисел.

Также определим сумматор особого назначения, называемый *инкрементором* и предназначенный для добавления 1 к указанному числу. (Названия «полусумматор» и «полный сумматор» основаны на особенностях реализации — полный сумматор можно реализовать с помощью двух полусумматоров, как мы увидим позже в этой главе.)

Полусумматор: первым шагом на пути к сложению двоичных чисел станет сложение двух битов. Будем рассматривать результат этой операции как 2-битное число и называть его правый и левый биты *суммой* и *переносом* соответственно. На иллюстрации 2.2 показан чип, выполняющий эту операцию сложения.



Chip name: HalfAdder
Input: a, b
Output: sum, carry
Function: sum \ LSB of a + b
 carry \ MSB of a + b

Иллюстрация 2.2. Полусумматор, предназначенный для сложения 2 битов.

Полный сумматор: на иллюстрации 2.3 показан чип *полного сумматора*, предназначенный для сложения трех битов. Как и в случае с полусумматором, полный сумматор выводит два бита, которые, взятые вместе, представляют собой сложение трех входных битов.

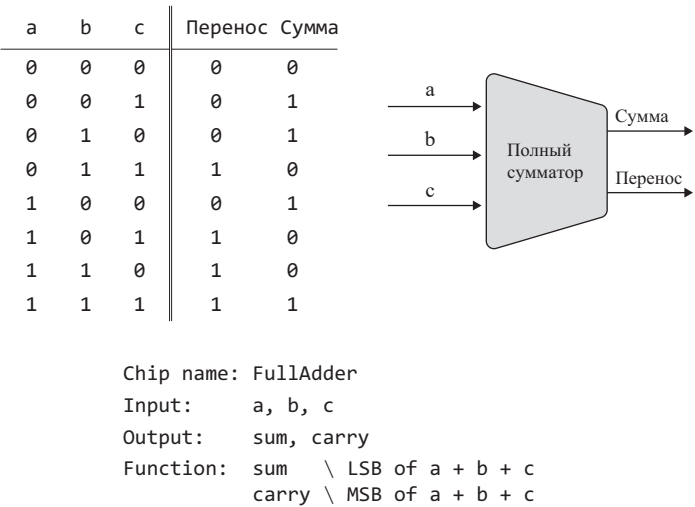


Иллюстрация 2.3. Полный сумматор, предназначенный для сложения 3 битов.

Сумматор: целые числа представлены в компьютерах в виде слова фиксированного размера — длиной в 8, 16, 32 или 64 бита. Микросхема, осуществляющая сложение двух таких *n*-битных чисел, называется сумматором. На иллюстрации 2.4 показан 16-битный сумматор.

Заметим мимоходом, что логическую схему для сложения 16-битных чисел легко можно расширить для реализации любой *n*-битной микросхемы сумматора, независимо от *n*.

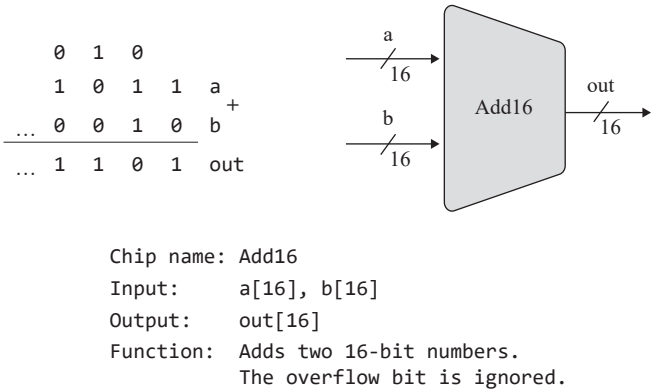


Иллюстрация 2.4. 16-битный сумматор, предназначенный для сложения двух 16-битных чисел, с примером сложения (слева).

Инкрементор: когда мы позже будем проектировать архитектуру компьютера, нам понадобится микросхема, которая прибавляет 1 к заданному числу (*спойлер*: это позволит извлекать из памяти следующую команду после выполнения текущей). Хотя операцию $x + 1$ можно реализовать и с помощью нашего сумматора общего назначения, более эффективно это сделает специальная микросхема *инкрементора*. Вот интерфейс микросхемы:

```
Chip name: Inc16
Input: in[16]
Output: out[16]
Function: out = in + 1
Comment: Бит переноса игнорируется
```

2.5.2. Арифметико-логическое устройство

Все представленные до сих пор чипы сумматоров были микросхемами общего назначения: такие микросхемы так или иначе использует любой компьютер, выполняющий арифметические операции. Основываясь на них, перейдем к описанию блока *арифметико-логического устройства* — микросхемы, которая впоследствии станет вычислительным центром нашего процессора. В отличие от общих рассмотренных до сих пор вентилях и микросхем, конструкция АЛУ уникальна для компьютера, построенного в рамках программы «От Nand до “Тетриса”» и носящего название Наск. Тем не менее в основе АЛУ Наск лежат общие принципы проектирования, а знания, полученные в процессе ее построения, окажутся весьма полезными. Кроме того, наша архитектура АЛУ позволяет достичь большой функциональности с минимальным набором внутренних деталей. В этом отношении она представляет собой хороший пример эффективной и элегантной логической конструкции.

Как следует из его названия, арифметико-логическое устройство — это микросхема, предназначенная для выполнения набора арифметических и логических операций. То, *какие именно* операции должно выполнять АЛУ, — это проектное решение, вытекающее из соображений

экономической эффективности. В случае платформы Nask мы решили, что 1) АЛУ будет выполнять операции только над целыми числами (а не, например, арифметические операции над числами с плавающей запятой) и 2) АЛУ будет выполнять восемнадцать арифметико-логических функций, перечисленных на иллюстрации 2.5а.

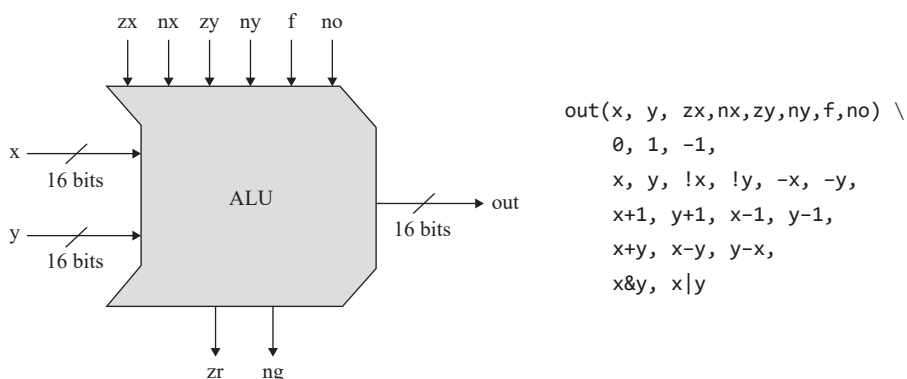


Иллюстрация 2.5а. АЛУ Nask, предназначенное для выполнения восемнадцати арифметико-логических функций, показанных справа (символы `!`, `&`, и `|` означают соответственно 16-битные операции Not, And и Or). Пока что не обращайтесь внимания на выходные биты *zr* и *ng*.

Как показано на иллюстрации 2.5а, АЛУ Nask работает с двумя 16-битными целыми числами, обозначаемыми *x* и *y*, и с шестью 1-битными входами, называемыми *управляющими (контрольными) битами*. Эти управляющие биты «сообщают» АЛУ, какую функцию вычислять. Точная спецификация приведена на рисунке 2.5б.

Для иллюстрации логики АЛУ предположим, что мы хотим вычислить функцию $x - 1$ для $x = 27$. Для начала подаем 16-битный двоичный код 27 на вход *x*. В данном конкретном примере значение *y* нас не волнует, так как оно не влияет на требуемое вычисление. После этого находим в таблице на иллюстрации 2.5б функцию $x - 1$ в столбце выхода (*out*) и устанавливаем управляющие биты АЛУ: 001110. Согласно спецификации, такая установка должна привести к тому, что АЛУ выдаст двоичный код, представляющий число 26. Так ли это? Чтобы выяснить, рассмотрим подробнее, как АЛУ Nask выполняет свою «магию». Посмотрите на верхнюю строку иллюстрации 2.5б и обратите

внимание, что каждый из шести управляющих битов соотносится с какой-то отдельной микрооперацией. Например, бит zx соотносится с «if ($zx == 1$) then set x to 0» («если ($zx == 1$), то установить x равным 0»). Эти шесть директив должны выполняться по порядку: сначала мы либо устанавливаем входы x и y на 0, либо нет; затем либо инвертируем полученные значения, либо нет; затем вычисляем + или & на предварительно обработанных значениях; и, наконец, либо инвертируем полученное значение, либо нет. Все эти установки, инвертирования, сложения и конъюнкции являются 16-битными операциями.

Предварительная настройка ввода x		Предварительная настройка ввода y		Вычисление + или &	Послеопера- ционная настройка вывода	Итоговый вывод АЛУ
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y) =
zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x y

Иллюстрация 2.56. Взятые вместе значения шести управляющих битов zx , nx , zy , ny , f и no заставляют АЛУ вычислять одну из функций, перечисленных в крайнем правом столбце.

Ориентируясь на данную логику, вернемся к строке, связанной с $x - 1$, и проверим, что микрооперации, закодированные шестью управляющими битами, действительно заставят АЛУ вычислить $x - 1$. Двигаясь слева направо, мы видим, что биты zx и px равны 0, поэтому не обнуляем и не инвертируем число на входе x , а оставляем его как есть. Биты zy и py равны 1, поэтому мы сначала обнуляем число на входе y , а затем инвертируем результат, получая 16-битное значение 1111111111111111. Так как этот двоичный код представляет собой -1 в дополнительном коде, мы видим, что на двух входах данных АЛУ находятся число x и -1 . Бит f равен 1, поэтому выполняется операция сложения, и АЛУ производит вычисление $x + (-1)$. Наконец, бит po равен 0, поэтому вывод не инвертируется. Итак, мы показали, что если подать на АЛУ некие значения x и y и установить шесть управляющих битов 001110, то АЛУ вычислит $x - 1$, как и требовалось.

Но что насчет остальных семнадцати функций, перечисленных на иллюстрации 2.5б? Действительно ли АЛУ вычисляет и их? Чтобы убедиться, что это действительно так, предлагаем вам внимательно рассмотреть другие строки таблицы, пройти через тот же процесс выполнения микроопераций, закодированных шестью управляющими битами, и выяснить, что же выдаст АЛУ на выходе. Или можете просто поверить нам, что АЛУ работает именно так, как и заявлено.

Обратите внимание, что АЛУ фактически вычисляет в общей сложности шестьдесят четыре функции, поскольку шесть управляющих битов кодируют именно такое количество возможностей. Мы решили сосредоточиться на некоторых из них и задокументировать только восемнадцать из этих возможностей, поскольку их будет достаточно для поддержки набора команд нашей целевой компьютерной системы. Любопытный читатель, вероятно, будет заинтригован, узнав, что некоторые из незадокументированных операций АЛУ весьма даже значимы. Однако мы решили не использовать их в системе Hack.

Интерфейс АЛУ Hack показан на рисунке 2.5в. Обратите внимание, что, помимо выполнения заданной функции над числами на своих двух входах, АЛУ также вычисляет два выходных бита zr и ng . Эти биты, которые отмечают, является ли выход АЛУ нулевым или

отрицательным соответственно, в дальнейшем будут использоваться центральным процессором нашей будущей компьютерной системы.

```

Chip name: ALU
Input:  x[16], y[16],    // Two 16-bit data inputs (два вводимых 16-битных числа)
        zx,             // Zero the x input (обнуление введенного x)
        nx,             // Negate the x input (инверсия введенного x)
        zy,             // Zero the y input (обнуление введенного y)
        ny,             // Negate the y input (инверсия введенного y)
        f,              // if f==1 out=add(x, y) else out=and(x, y) (если f
                        // равно 0, производится сложение x и y, иначе операция
                        // логического И над x и y)
        no              // Negate the out output (инвертирование числа,
                        // направляемого на вывод)
Output: out[16],        // 16-bit output(вывод 16-битного числа)
        zr,             // if out==0 zr=1 else zr=0 (если число на выходе
                        // равно 0, zr=1, иначе zr=0)
        ng              // if out<0 ng=1 else ng=0 (если число на выходе
                        // отрицательное, то ng=1, иначе ng=0)

Function:
    if zx x=0           // 16-bit zero constant (16-битная константа ноль)
    if nx x=!x          // Bit-wise negation (побитовая инверсия)
    if zy y=0           // 16-bit zero constant (16-битная константа ноль)
    if ny y=!y          // Bit-wise negation (побитовая инверсия)
    if f out=x+y        // Integer two's complement addition (побитовое сложение
                        // с дополнительным кодом)
    else out=x&y        // Bit-wise And (побитовое И)
    if no out=!out      // Bit-wise negation (побитовая инверсия)
    if out==0 zr=1 else zr=0 // 16-bit equality comparison (16-битное
                        // сравнение)
    if out<0 ng=1 else ng=0 // two's complement comparison (сравнение
                        // дополнительного кода)
Comment: The overflow bit is ignored (бит переполнения игнорируется)

```

Иллюстрация 2.5в. API АЛУ Hack.

Возможно, будет поучительно описать ход мыслей, который привел к разработке нашего АЛУ. Во-первых, мы составили предварительный список примитивных операций, которые должен был выполнять наш компьютер (правая колонка иллюстрации 2.5б). Затем постарались рассуждать в обратном направлении и выяснить, как можно манипулировать двоичными числами x , y и out для выполнения желаемых операций. Эти требования к обработке чисел, а также наша цель максимально упростить логику АЛУ привели к решению использовать шесть управляющих битов, каждый из которых связан с простой операцией, которую можно легко реализовать с помощью базовых

логических вентилях. В результате АЛУ получилось простым и элегантным. А в аппаратном обеспечении простота и элегантность — это далеко не самые последние качества.

2.6. Реализация

Наши рекомендации по реализации намеренно минимальны. Мы уже дали много советов на эту тему, и теперь ваша очередь обнаружить недостающие части в архитектурах микросхем.

Во всем этом разделе, когда мы говорим «постройте/реализуйте логическую схему, которая...», мы ожидаем, что вы: 1) составите план логической схемы (например, нарисовав расположение вентилях), 2) напишете HDL-код, реализующий эту схему, и 3) протестируете и отладите свою схему с помощью прилагаемых тестовых скриптов и симулятора аппаратуры. Более подробная информация приведена в следующем разделе, где описывается проект 2.

Полусумматор: таблица истинности на иллюстрации 2.2 показывает, что выходы `sum (a, b)` и `carry (a, b)` совпадают с выходами двух простых булевых функций, рассмотренных и реализованных в проекте 1. Таким образом, реализация полусумматора довольно очевидна.

Полный сумматор: чип полного сумматора можно реализовать из двух полусумматоров и одного дополнительного вентиля (именно поэтому такие сумматоры называются *половинным* и *полным*). Возможны и другие варианты реализации, в том числе и непосредственные, без использования полусумматоров.

Сумматор: сложение двух n -битных чисел можно выполнять побитно справа налево. На шаге 0 складывается самая младшая пара битов, а полученный бит переноса подается на сложение следующей значащей пары битов. Процесс продолжается до тех пор, пока не будет сложена пара самых старших значащих битов. Обратите внимание, что

на каждом шаге происходит сложение трех битов, один из которых переносится из «предыдущего» сложения.

Читатели могут задаться вопросом, как же можно складывать пары битов «параллельно», до того как бит переноса будет вычислен предыдущей парой. Ответ заключается в том, что эти вычисления выполняются достаточно быстро, чтобы завершиться и стабилизироваться в течение одного тактового цикла. Мы обсудим тактовые циклы и синхронизацию в следующей главе, а пока можно полностью игнорировать элемент времени и написать HDL-код, который вычисляет операцию сложения, как если бы все пары битов складывались одновременно.

Инкрементор: любой n -битный инкрементор можно легко реализовать множеством разных способов.

АЛУ: наше АЛУ было тщательно спланировано, чтобы все желаемые операции выполнялись *логически*, с использованием простых булевых операций, подразумеваемых шестью управляющими битами. Поэтому физическую реализацию АЛУ можно свести к выполнению этих простых операций, следуя спецификациям псевдокода в верхней части иллюстрации 2.56. Вашим первым шагом, скорее всего, будет создание логической схемы для обнуления и инверсии (отрицания, операции НЕ) 16-битного значения. Эту логику можно использовать для обработки входных значений x и y , а также выходного значения out . Микросхемы для побитовой операции And (И) и побитового сложения уже были созданы в проектах 1 и 2 соответственно. Таким образом, осталось построить логику, которая выбирает между этими двумя операциями в соответствии с управляющим битом f (логика выбора также была реализована в проекте 1). Как только эта основная функциональность АЛУ заработает как следует, можно будет перейти к реализации необходимой функциональности выводов однобитных значений zr и ng .

2.7. Проект

Задача: реализовать все микросхемы, представленные в данной главе. Единственные строительные блоки, которые вам понадобятся, — это некоторые вентили, описанные в главе 1, и микросхемы, которые вы будете постепенно собирать в текущем проекте.

Встроенные чипы: как только что было сказано, микросхемы, которые вы будете собирать в этом проекте, используют в качестве составных частей некоторые из микросхем, описанных в главе 1. Даже если вы успешно построили микросхемы нижнего уровня на HDL, мы рекомендуем использовать их встроенные версии. Вот вам полезный совет для всех проектов курса «От Nand до “Тетриса”»: всегда предпочитайте использовать встроенные микросхемы вместо их HDL-реализаций. Встроенные микросхемы гарантированно работают в соответствии со спецификацией и предназначены для ускорения работы симулятора аппаратуры.

Следовать этому совету очень просто: не добавляйте в папку проекта `nand2tetris/projects/02` ни одного `.hdl` файла из проекта 1. Всякий раз, когда аппаратный симулятор встретит в вашем HDL-коде ссылку на часть микросхемы из проекта 1, например `And16`, он будет проверять, есть ли в текущей папке файл `And16.hdl`. Если такой файл найден не будет, симулятор аппаратуры по умолчанию воспользуется встроенной версией данной микросхемы, что вам и требуется.

Остальные рекомендации для этого проекта такие же, как и для проекта 1. В частности, помните, что в хороших HDL-программах используется как можно меньше составных частей. Нет необходимости придумывать и реализовывать какие-то свои «вспомогательные микросхемы»; в ваших HDL-программах должны применяться только те микросхемы, которые были описаны в главах 1 и 2.

Веб-версия проекта 2 доступна на сайте www.nand2tetris.org.

2.8. Перспектива

В этой главе была описана стандартная конструкция многоразрядного сумматора, хотя и не уделялось никакого внимания ее эффективности. И действительно, предложенная нами реализация сумматора неэффективна из-за задержек, возникающих при распространении битов переноса по n -разрядным слагаемым. Эти вычисления можно ускорить с помощью логических схем, использующих так называемую эвристику *ускоренного переноса*. Поскольку сложение — это наиболее распространенная операция в компьютерных архитектурах, любое такое низкоуровневое улучшение может привести к значительному росту производительности всей системы. Однако в данной книге мы сосредоточимся в основном на функциональности, оставив тему оптимизации микросхем для более специализированных книг и курсов по аппаратному обеспечению*.

Общая функциональность любой аппаратно-программной системы обеспечивается совместно центральным процессором и операционной системой, которая работает поверх аппаратной платформы. Таким образом, при проектировании новой компьютерной системы вопрос о том, как распределить желаемую функциональность между АЛУ и ОС, по сути, становится дилеммой выбора между стоимостью и производительностью. Как правило, прямые аппаратные реализации арифметических и логических операций более эффективны, чем программные реализации, но делают аппаратную платформу более дорогой.

Компромисс, который мы выбрали в программе «От Nand до “Тетриса”», заключается в разработке базового АЛУ с минимальной функциональностью и в использовании системного программного обеспечения для реализации дополнительных математических операций по мере необходимости. Например, в нашем АЛУ нет ни умножения,

* Техническая причина отказа от использования методов ускоренного переноса в наших микросхемах сумматоров заключается в том, что их аппаратная реализация требует циклических соединений выводов, которые не поддерживаются симулятором аппаратуры «От Nand до “Тетриса”».

ни деления. Во второй части книги, где будет дано обсуждение операционной системы (глава 12), мы реализуем элегантные и эффективные побитовые алгоритмы для умножения и деления, а также других математических операций. Эти процедуры ОС будут доступны для использования компиляторами языков высокого уровня, работающих поверх платформы Наск. Таким образом, когда программист на высокоуровневом языке записывает, допустим, выражение $x * 12 + \text{sqrt}(y)$, то после компиляции некоторые части выражения будут высчитываться непосредственно АЛУ, а некоторые — ОС, но программист высокого уровня совершенно не будет замечать этого низкоуровневого разделения работы. И действительно, одна из ключевых ролей операционной системы заключается в устранении разрывов между используемыми программистами абстракциями языка высокого уровня и аппаратным обеспечением, на котором эти абстракции реализуются.

3. Память

Неважная это память, если она работает только в одну сторону.

— Льюис Кэрролл (1832–1989)

Рассмотрим высокоуровневую операцию $x = y + 17$. В главе 2 мы показали, как можно использовать логические вентили для представления чисел и вычисления простых арифметических выражений, таких как $y + 17$. Теперь мы обсудим, как можно использовать логические вентили для *хранения значений на какое-то время* — в частности, что переменной x можно задать некое значение и что она будет хранить его до тех пор, пока мы не зададим ей другое значение. Для этого разработаем новое семейство *микросхем памяти*.

До сих пор все микросхемы, которые мы собирали в главах 1 и 2, вплоть до довольно сложного АЛУ, были независимы от времени. Такие микросхемы иногда называют *комбинационными*: они реагируют на различные комбинации своих входов без задержки, за исключением времени, которое требуется внутренним частям микросхемы для завершения вычислений. В этой главе мы опишем и построим *последовательные (последовательностные)* микросхемы. В отличие от комбинационных, которые не зависят от времени, сигналы на выходах последовательных микросхем зависят не только от сигналов на входах в текущий момент времени, но и от сигналов на входах и выходах, которые были обработаны ранее.

Само собой разумеется, что понятия «текущий» и «ранее» идут рука об руку с понятием времени: сейчас вы помните то, что было

зафиксировано в вашей памяти раньше. Таким образом, прежде чем рассуждать о памяти, нужно сначала выяснить, как использовать логику для моделирования хода времени. Это можно сделать с помощью своего рода *часов* (генератора тактовой частоты), генерирующих непрерывный ряд двоичных сигналов, которые мы называем «тик» и «так». Время между началом «тика» и концом последующего «така» называется *циклом*, и эти циклы будут использоваться для регулирования работы всех используемых в компьютере микросхем памяти.

После краткого, ориентированного на пользователя введения в устройства памяти мы опишем секвенциальную (последовательностную) логику, которой будем придерживаться при построении микросхем, зависящих от времени. Затем приступим к построению регистров, устройств оперативной памяти и счетчиков. Эти устройства памяти вместе с арифметическими устройствами, построенными в главе 2, охватывают все множество микросхем, необходимых для создания полной компьютерной системы общего назначения — задачи, которую мы будем решать в главе 5.

3.1. Устройства памяти

Компьютерные программы оперируют переменными, массивами и объектами — абстракциями, сохраняющими данные во времени. Аппаратные платформы поддерживают эту способность, предлагая устройства памяти, умеющие *сохранять свое состояние*. Поскольку эволюция наделила человека феноменальной системой электрохимической памяти, мы склонны считать само собой разумеющейся способность запоминать нечто, то есть удерживать информацию с течением времени. Однако такое свойство трудно реализовать в классической логике, которая не знает ни времени, ни состояния. Таким образом, чтобы продолжить работу, мы должны сначала найти способ смоделировать течение времени и наделить логические вентили способностью сохранять состояние и реагировать на изменения времени.

Подойдем к решению данной задачи, введя своеобразные «часы» (*генератор тактовых импульсов*) и элементарный, зависящий

от времени логический вентиль, способный переключаться между двумя стабильными состояниями: представлять 0 и представлять 1. Этот вентиль, называемый триггером (по-английски он называется *data flip-flop, DFF*), послужит нам фундаментальным строительным блоком, из которого будут построены все устройства памяти. Несмотря на свою центральную роль, триггер — довольно неприметный вентиль: в отличие от регистров, устройств оперативной памяти и счетчиков, играющих заметную роль в компьютерных архитектурах, триггеры используются неявно, как низкоуровневые микросхемы, встроенные глубоко в другие устройства памяти.

Фундаментальная роль триггера (DFF) хорошо видна на иллюстрации 3.1, где он служит основой иерархии памяти, которую мы собираемся построить. Мы покажем, как триггеры могут использоваться для создания 1-битных регистров и как n таких регистров можно соединить вместе для создания n -битного регистра. Далее построим устройство ОЗУ (*оперативное запоминающее устройство*, или RAM, *random access memory*, «память с произвольным доступом»), содержащее произвольное количество таких регистров. Помимо прочего, мы разработаем средство *адресации*, то есть мгновенного и непосредственного обращения к любому произвольно выбранному регистру из ОЗУ.

Однако прежде чем приступить к созданию этих микросхем, мы представим методологию и инструменты, позволяющие моделировать течение времени и поддержание состояния во времени.

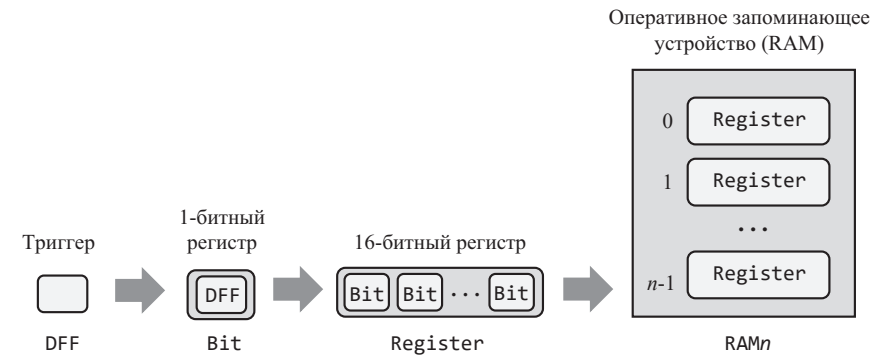


Иллюстрация 3.1. Иерархия памяти, которая будет построена в данной главе.

3.2. Секвенциальная логика

Все микросхемы, рассмотренные в главах 1 и 2, были основаны на классической логике, не зависящей от времени. Для того чтобы разработать устройства памяти, нужно дополнить нашу логику вентилей способностью реагировать не только на изменения входных данных, но и на регулярную смену импульсов («тиканье часов»): например, мы помним значение слова «собака» в момент времени t , поскольку помнили его в момент времени $t - 1$, и так вплоть до того момента, когда впервые записали его в свою память. Чтобы наделить компьютер такой способностью поддерживать состояние со временем, мы должны расширить нашу компьютерную архитектуру временным измерением и создать инструменты, которые работают со временем с помощью булевых функций.

3.2.1. Вопросы времени

До сих пор в нашем путешествии «От Nand до “Тетриса”» мы предполагали, что микросхемы реагируют на ввод мгновенно: допустим, мы вводим числа 7 и 2, подаем АЛУ команду «вычесть» и... *готово!* АЛУ выдает число 5. В реальности же вывод всегда запаздывает — по крайней мере, по двум причинам. Во-первых, значения на входах микросхем не появляются из воздуха; представляющие их сигналы, скорее всего, поступают с выходов других микросхем, и их перемещение занимает некоторое время. Во-вторых, вычисления, которые выполняют микросхемы, также требуют времени — чем больше в микросхеме чистей, тем сложнее ее логика и тем больше времени требуется для формирования значения, подаваемого на ее выходы.

Таким образом, в процессе создания компьютера неизбежно всплывают вопросы времени, которые необходимо решать. Как показано в верхней части рисунка 3.2, время обычно рассматривается в виде метафорической стрелы, неустанно движущейся вперед. Это движение считается непрерывным: между каждыми двумя точками времени можно выделить еще одну временную точку, а изменения в реальном

мире бывают бесконечно малыми. Такое представление о времени, популярное среди философов и физиков, слишком глубокое и загадочное для компьютерных инженеров. Поэтому вместо того, чтобы рассматривать время как непрерывную прогрессию, мы предпочитаем разбивать его на интервалы фиксированной длины, называемые *циклами*. Такое представление называется дискретным, в результате чего получаются цикл 1, цикл 2, цикл 3 и т. д. В отличие от непрерывной стрелы времени, которую можно делить на сколь угодно малые отрезки, циклы атомарны и неделимы: какие-то изменения в мире происходят только во время переходов между циклами; внутри же циклов мир стоит на месте.

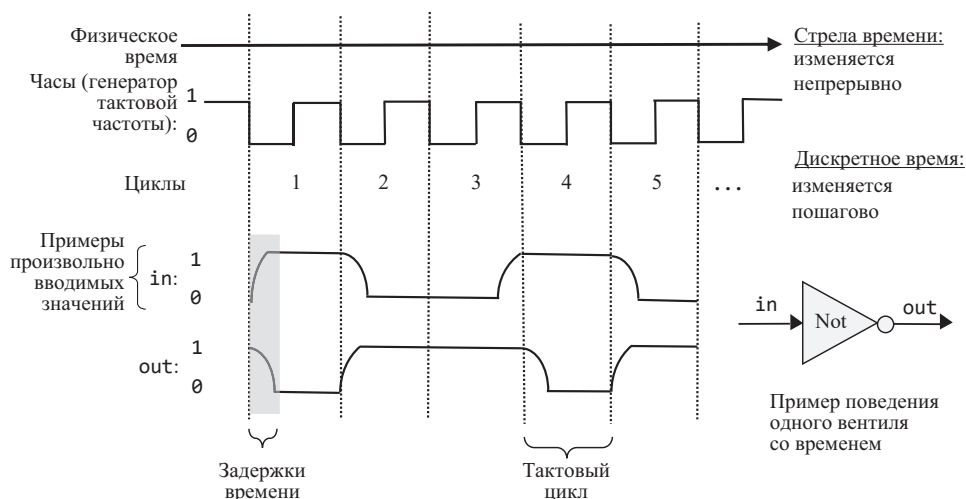


Иллюстрация 3.2. Дискретное представление времени. Изменения состояния (входных и выходных значений) наблюдаются только при переходах цикла. Внутри циклов изменения игнорируются.

Конечно же, в реальности мир никогда не стоит на месте. Однако рассматривая время дискретно, мы сознательно и намеренно игнорируем непрерывные изменения. Мы довольствуемся тем, что знаем состояние мира в цикле n , а затем в цикле $n + 1$, но каково состояние мира в течение каждого цикла — нам это все равно. Что касается создания компьютерных архитектур, то такой дискретный взгляд

на время служит двум важным целям проектирования. Во-первых, так нейтрализуется влияние случайности, связанной с задержками связи и вычислений. Во-вторых, как мы увидим позже, таким образом данный подход ко времени позволяет синхронизировать операции различных микросхем во всей системе.

В качестве пояснения рассмотрим нижнюю часть иллюстрации 3.2, где показано, как вентиль Not (используемый для примера) реагирует на значения, произвольно подаваемые на его вход. Когда мы подаем значение 1, проходит некоторое время, прежде чем значение на выходе стабилизируется и становится равным 0. Но поскольку длительность цикла намеренно больше временной задержки, то к концу цикла на выходе мы всегда получаем значение 0. Поскольку мы проверяем состояние мира только в конце цикла, то не замечаем промежуточных временных задержек; нам кажется, будто на вентиль подан сигнал 0 и он тут же — бац! — выдал сигнал 1. Если обобщить наблюдения, то можно сделать вывод, что всякий раз, когда на вентиль Not подается некоторый двоичный вход x , этот вентиль мгновенно выдает код Not (x).

Проницательные читатели, вероятно, уже заметили, что для работы данной схемы *продолжительность* (или *период цикла*) должна быть больше максимальных временных задержек, которые могут возникнуть в системе. И действительно, период цикла (или же *тактовая частота* — количество циклов в единицу времени) — это один из важнейших параметров проектирования любой аппаратной платформы: при проектировании компьютера инженер по аппаратному обеспечению выбирает период цикла, отвечающий двум целям проектирования. С одной стороны, цикл должен быть достаточно длинным, чтобы вместить в себя и нейтрализовать любую возможную временную задержку; с другой стороны, чем короче цикл, тем быстрее работает компьютер: если события происходят только во время переходов цикла, то очевидно, что они происходят быстрее, когда циклы короче. В целом выбирается такой период цикла, чтобы он был немного больше максимальной временной задержки в любом чипе системы. Благодаря огромному прогрессу в технологиях коммутации, в настоящее

время мы можем поддерживать циклы в миллиардную долю секунды и достигать тем самым поразительной скорости работы компьютеров.

Обычно циклы реализуются осциллятором, который непрерывно чередует две фазы, обозначаемые 0 — 1, *низкий* — *высокий* сигнал, или «тик-так» (как показано на иллюстрации 3.2). Время между началом «тика» и концом последующего «така» называется *циклом*, и каждый цикл моделирует одну дискретную единицу времени. Текущая фаза цикла («тик» или «так») представлена двоичным сигналом. С помощью аппаратной схемы один и тот же главный тактовый сигнал одновременно транслируется на все микросхемы памяти в системе. Тактовый вход каждой такой микросхемы соединен с вентилями-триггерами нижнего уровня, благодаря чему микросхема фиксирует новое состояние и выводит его только в конце тактового цикла.

3.2.2. Триггеры

Микросхемы памяти предназначены для «запоминания» (или хранения) информации с течением времени. Низкоуровневые устройства, реализующие эту абстракцию хранения информации, называются триггерами, и их бывает несколько типов. В «От Nand до “Тетриса”» мы используем вариант под названием «триггер данных» (*data flip-flop, DFF*), интерфейс которого включает однобитный вход данных и однобитный выход данных (см. верхнюю часть иллюстрации 3.3).

Кроме того, DFF имеет тактовый вход, который питается от генератора тактовой частоты. Вместе взятые, вход данных и тактовый вход позволяют DFF реализовать простое поведение с учетом времени $out(t) = in(t - 1)$, где *in* и *out* — входные и выходные значения вентиля, а *t* — текущая единица времени (в дальнейшем мы будем использовать термины «единица времени» и «цикл» как взаимозаменяемые). Пусть нас не волнует, как на самом деле реализуется такое поведение. Пока мы просто наблюдаем, что в конце каждой единицы времени DFF выводит на вход значение, полученное в предыдущую единицу времени.

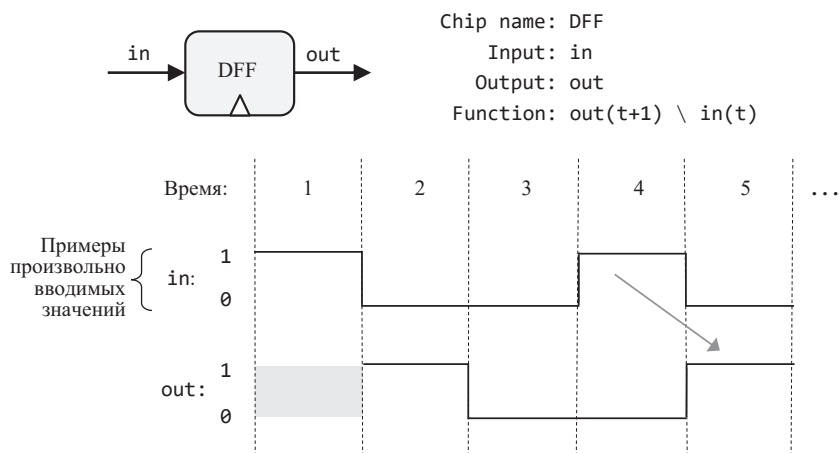


Иллюстрация 3.3. Триггер данных (вверху) и пример его поведения (внизу). В первом цикле предыдущее значение на входе неизвестно, поэтому выход DFF не определен. В каждую последующую единицу времени DFF выводит значение, которое было на входе в предыдущую единицу времени. В соответствии с правилами отображения вентилях на схемах вход тактового генератора обозначен маленьким треугольником, нарисованным в нижней части пиктограммы вентиля.

Как и вентили Nand, вентили DFF находятся на самых нижних уровнях аппаратной иерархии. Как показано на иллюстрации 3.1, на вентильх DFF в конечном счете основаны все микросхемы памяти в компьютере: регистры, блоки оперативной памяти и счетчики. Все их DFF подключены к одному и тому же генератору тактовой частоты, образуя своеобразный «кордебалет», выполняющий синхронные операции. В конце каждого тактового цикла выходы всех DFF в компьютере фиксируют значения, которые были на их входах в предыдущем цикле. Все остальное время DFF, как говорится, «заперты», то есть заблокированы от изменений — любые подаваемые на вход изменения не оказывают немедленного воздействия на их выходы. Все многочисленные DFF-вентили системы повторяют эту операцию по фиксации значений множество раз в секунду (в зависимости от тактовой частоты компьютера).

Аппаратная реализация зависимости от времени осуществляется с помощью выделенной тактовой шины, подающей сигнал генератора

тактовой частоты одновременно на все DFF-вентили системы. Симуляторы аппаратного обеспечения эмулируют тот же эффект в программном обеспечении. В частности, в симуляторе аппаратуры «От Nand до “Тетриса”» есть значок часов, позволяющий пользователю интерактивно переводить часы, а также программные команды «tick» и «tock», которые можно использовать в сценариях тестирования.

3.2.3. Комбинационная и секвенциальная логики

Все микросхемы, разработанные в главах 1 и 2 (начиная с элементарных логических вентилей и заканчивая АЛУ), были спроектированы так, чтобы реагировать только на изменения, происходящие в течение текущего тактового цикла. Такие микросхемы называются *независимыми от времени*, или *комбинационными*. Последнее название связано с тем, что данные микросхемы реагируют только на различные комбинации своих входных значений без учета хода времени.

В отличие от них, микросхемы, предназначенные для реагирования на изменения, произошедшие в предыдущие единицы времени (и, возможно, в текущую единицу времени), называются тактовыми, последовательными, или *последовательностными*. Наиболее фундаментальный последовательный вентиль — это триггер DFF, и любая микросхема, прямо или косвенно включающая данный триггер, также называется последовательностной. На иллюстрации 3.4 изображена типичная конфигурация последовательностной, или *секвенциальной* логики. Основным элемент такой конфигурации — одна или несколько микросхем, в состав которых прямо или косвенно входят триггеры DFF. Как показано на иллюстрации, эти последовательностные микросхемы могут также взаимодействовать с комбинационными микросхемами. Контур обратной связи позволяет последовательностной микросхеме реагировать на значения входов и выходов, какими они были в предыдущую единицу времени. В комбинационных микросхемах, где время не моделируется и не учитывается, реализовать обратную связь проблематично; значение на выходе чипа будет зависеть от значения на его входе, который сам будет зависеть от выхода и, таким образом, сам от себя. Однако если контур обратной связи проходит через

вентиль DFF, то никакой сложности в том, чтобы подавать значение выхода обратно на вход, нет: DFF выполняет временную задержку, так что значение на выходе в момент времени t зависит не от самого себя, а от значения на выходе в момент времени $t - 1$.

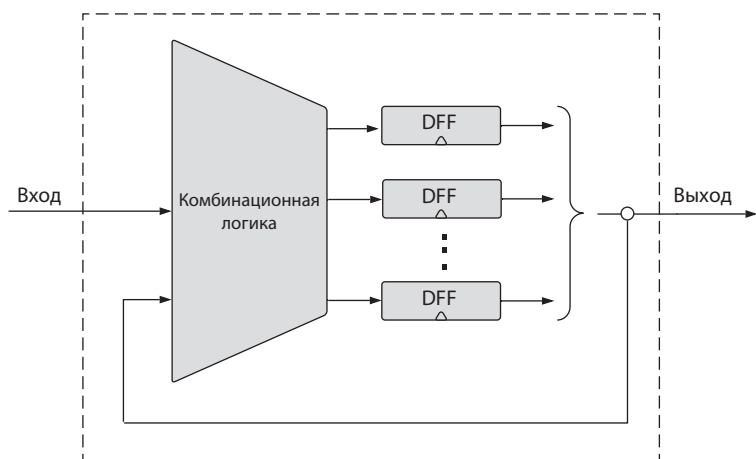


Иллюстрация 3.4. Дизайн секвенциальной логики обычно подразумевает наличие вентилях DFF, связанных с комбинационными микросхемами и получающих данные от них. Это дает последовательным микросхемам возможность реагировать как на текущие, так и на предыдущие значения входов и выходов.

Зависимость от времени, присущая последовательным микросхемам, имеет важный побочный эффект, служащий для синхронизации общей архитектуры компьютера. Для иллюстрации предположим, что мы даем команду АЛУ вычислить выражение $x + y$, где x — значение на выходе расположенного поблизости регистра, а y — значение на выходе удаленного регистра ОЗУ. Из-за физических ограничений — таких как расстояние, сопротивление и помехи — электрические сигналы, представляющие x и y , скорее всего, придут в АЛУ в разное время. Однако АЛУ, будучи комбинационной микросхемой, нечувствительно к понятию времени, оно непрерывно складывает любые данные, которые оказываются на его входах. Таким образом, должно пройти некоторое время, прежде чем на выходе АЛУ

зафиксируется правильный результат $x + y$. До тех пор АЛУ будет генерировать мусор.

Как же решить данную проблему? С дискретным представлением времени это легко, она *даже не должна нас заботить*. Все, что нужно сделать, — установить тактовую частоту генератора таким образом, чтобы период цикла был немного больше, чем время, которое требуется биту на прохождение самого длинного расстояния от одной микросхемы до другой, плюс время, необходимое для завершения самого трудоемкого вычисления внутри микросхемы. Таким образом, мы гарантируем, что к концу тактового цикла выходной сигнал АЛУ будет правильным. Иначе говоря, это трюк, превращающий набор отдельных аппаратных компонентов в хорошо синхронизированную систему. Подробнее о такой синхронной организации мы расскажем, когда будем строить архитектуру компьютера в главе 5.

3.3. Спецификация

Перейдем теперь к спецификации микросхем памяти, которые обычно используются в компьютерных архитектурах:

- триггеры данных (DFF);
- регистры (на основе DFF);
- устройства оперативной памяти ОЗУ (на основе регистров);
- счетчики (на основе регистров).

Как обычно, опишем эти микросхемы абстрактно. В частности, сосредоточимся на интерфейсе каждой микросхемы: ее входах, выходах и функции. О том, как микросхемы обеспечивают такую функциональность, будет сказано в разделе «Реализация».

3.3.1. Триггер данных

Самое элементарное последовательное устройство, которое мы будем использовать, и основной компонент, из которого будут построены все остальные микросхемы памяти, — это триггер данных (DFF).

DFF-вентиль имеет однобитный вход данных, однобитный выход данных, тактовый вход и простое поведение, зависящее от времени: $out(t) = in(t - 1)$.

Использование: если на вход DFF подать однобитное значение, то состояние DFF будет установлено на это значение, и оно будет представлено на выходе DFF в следующую единицу времени (см. рисунок 3.3). Эта, казалось бы, скромная операция окажется наиболее полезной при описанной далее реализации регистров.

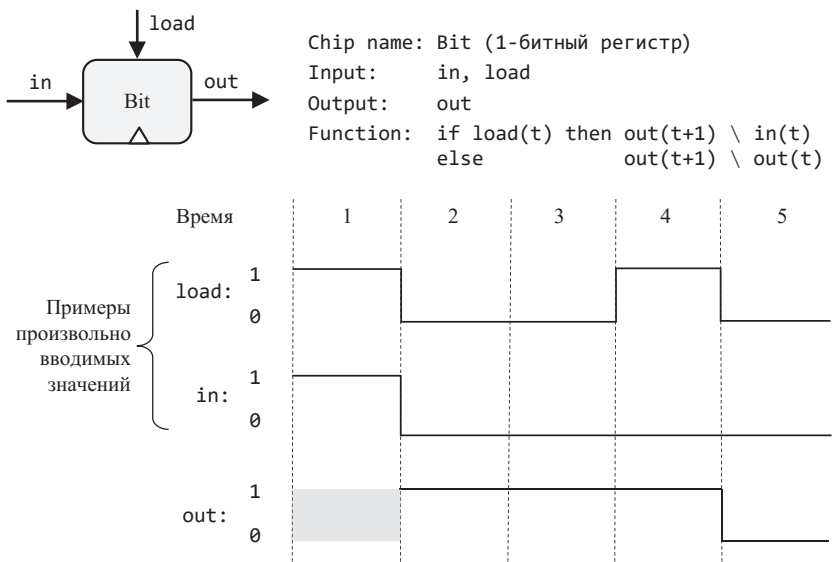


Иллюстрация 3.5. 1-битный регистр. Хранит и выдает 1-битное значение, пока не получит команду записать новое значение.

3.3.2. Регистры

Мы представляем здесь однобитный регистр под названием Bit и 16-битный регистр под названием Register. Микросхема Bit предназначена для хранения одного бита информации (0 или 1) на протяжении времени. Интерфейс микросхемы состоит из входа in, содержащего бит данных, входа load, включающего регистр для записи, и выхода out, выдающего текущее состояние регистра. API

регистра Bit и поведение его входов/выхода описаны на иллюстрации 3.5.

На иллюстрации 3.5 показано поведение однобитного регистра во времени, реагирующего на произвольные примеры значений входов in и load. Обратите внимание, что, независимо от входного значения, до тех пор, пока бит load не принял значение 1, регистр заблокирован и сохраняет свое текущее состояние.

Микросхема 16-битного регистра Register ведет себя точно так же, как и микросхема Bit, за исключением того, что она предназначена для работы с 16-битными значениями. Подробности приведены на иллюстрации 3.6.

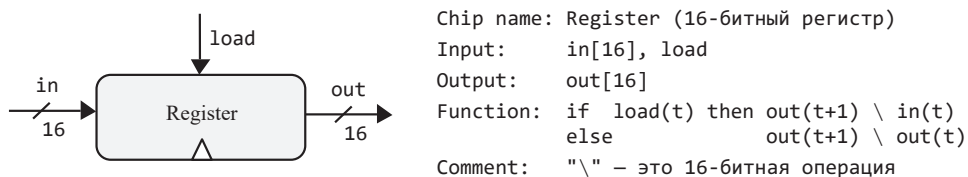


Иллюстрация 3.6. 16-битный регистр. Хранит и выдает 16-битное значение, пока не получит команду записать новое значение.

Использование: однобитный регистр Bit и 16-битный регистр Register используются одинаково. Чтобы прочитать состояние регистра, обратитесь к значению out. Чтобы установить состояние регистра v , подайте v на вход in и активизируйте вход load (установите его значение равным 1). Так текущим состоянием регистра станет v , и начиная со следующей единицы времени на его выходе будет зафиксировано это новое значение. Видно, что микросхема Register выполняет классическую функцию запоминающего устройства: она запоминает и выдает последнее значение, которое было в нее записано, до тех пор, пока не будет записано новое значение.

3.3.3. Оперативное запоминающее устройство

Блок памяти с прямым доступом, иначе говоря, запоминающее устройство с произвольным доступом, или *оперативное запоминающее*

устройство (ОЗУ) — это устройство, собранное на основе n количества микросхем Register. Указав определенный адрес (число от 0 до $n - 1$), можно выбрать любой регистр оперативной памяти и сделать его доступным для операций чтения/записи. Важно отметить, что доступ к любому произвольно выбранному регистру памяти осуществляется мгновенно и не зависит от адреса регистра и размера ОЗУ. Именно это делает устройства ОЗУ такими удивительно полезными: даже если они содержат миллиарды регистров, мы все равно можем получить доступ и манипулировать каждым выбранным регистром напрямую и мгновенно. API ОЗУ показан на иллюстрации 3.7.

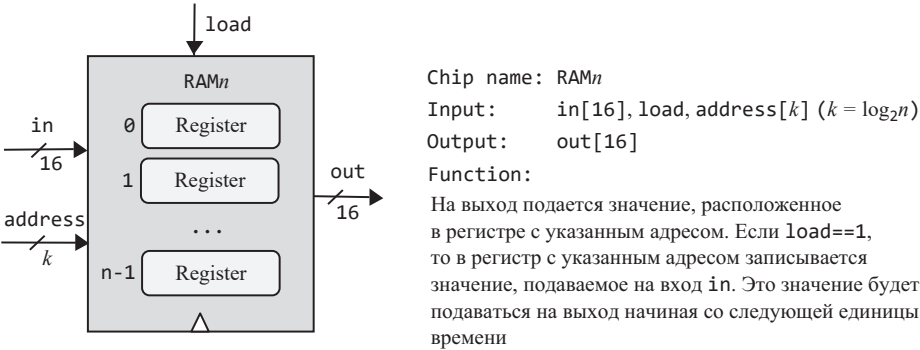


Иллюстрация 3.7. Микросхема ОЗУ, состоящая из n 16-битных чипов Register, которые можно выбирать и с которыми можно работать по отдельности. Адреса регистров (от 0 до $n - 1$) не являются аппаратной частью микросхемы. Они реализуются с помощью логики, которая будет обсуждаться в разделе «Реализация».

Использование: чтобы прочитать содержимое регистра номер m , подайте на вход address значение m . Это действие выберет регистр номер m , и на выходе out ОЗУ появится его значение. Чтобы записать новое значение v в регистр номер m , подайте на вход address значение m , на вход in значение v и активизируйте вход load (установите его значение равным 1). Это действие выберет регистр номер m , разрешит его для записи и запишет в него значение v . В следующую единицу времени этот регистр ОЗУ станет подавать на выход значение v .

В итоге устройство ОЗУ ведет себя именно так, как и было задумано: как банк адресуемых регистров, к каждому из которых можно обращаться и с каждым из которых можно работать независимо. В случае операции чтения (`load==0`) на выходе ОЗУ немедленно появляется значение выбранного регистра. В случае операции записи (`load==1`) в выбранный регистр памяти записывается входное значение, и ОЗУ начинает выдавать его со следующей единицы времени.

Важно, что реализация оперативной памяти должна гарантировать почти мгновенное время доступа к любому регистру оперативной памяти. Если бы это было не так, мы не смогли бы извлекать команды и манипулировать переменными за разумное время, а компьютеры работали бы слишком медленно. «Магия» мгновенного времени доступа будет раскрыта в ближайшее время в разделе «Реализация».

3.3.4. Счетчик

Счетчик — это микросхема, способная увеличивать свое значение на 1 в каждую единицу времени. Когда мы будем строить архитектуру компьютера в главе 5, то будем называть данную микросхему *Program Counter* (*счетчиком команд* или *программным счетчиком*) или *PC*, поэтому аналогичное название будем использовать и здесь.

Интерфейс микросхемы *PC* идентичен интерфейсу регистра, за исключением того, что у нее есть управляющие биты, обозначенные `inc` и `reset`. При `inc==1` счетчик увеличивает свое состояние в каждом тактовом цикле, выполняя операцию `PC++`. Если мы хотим сбросить счетчик на 0, то активируем вход `reset` (подаем на него значение 1); если хотим установить счетчик на значение *v*, то подаем значение *v* на вход `in` и активируем вход `load` — как обычно поступаем с регистрами. Подробности приведены на иллюстрации 3.8.

Использование: чтобы прочитать текущее содержимое *PC*, обратитесь к значению вывода `out`. Чтобы сбросить *PC*, подайте сигнал 1 на бит `reset` и установите остальные управляющие биты на 0. Чтобы содержимое *PC* увеличивалось на 1 в каждую единицу

времени до дальнейшего уведомления, активируйте вход `inc` (подайте на него 1) и установите остальные управляющие биты на 0. Чтобы установить PC на значение v , подайте на вход `in` значение v , активируйте вход `load` и установите остальные управляющие биты на 0.

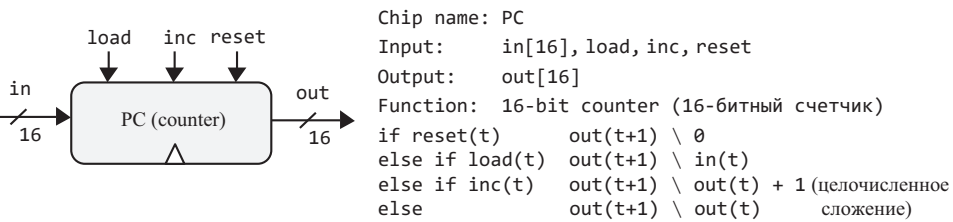


Иллюстрация 3.8. Счетчик Program Counter (PC). Для корректного его использования следует активировать (установить значение 1) по крайней мере один из управляющих битов `load`, `inc` или `reset`.

3.4. Реализация

В предыдущем разделе было представлено семейство абстракций микросхем памяти с упором на их интерфейс и функциональность. Этот раздел посвящен реализации микросхем с использованием уже построенных более простых чипов. Как обычно, наши рекомендации по реализации намеренно носят рекомендательный характер; мы хотим дать достаточно подсказок, чтобы вы могли завершить реализацию самостоятельно, используя HDL и прилагаемый симулятор аппаратуры.

3.4.1. Триггер данных (DFF)

Вентиль DFF предназначен для переключения между двумя стабильными состояниями 0 и 1. Эту функциональность можно реализовать несколькими различными способами, в том числе с использованием только вентилях Nand. Но, несмотря на свою элегантность, такие реализации DFF на основе Nand сложны. Их невозможно смоделировать в нашем симуляторе аппаратуры, поскольку они требуют контуров

обратной связи между комбинационными вентилями. Желая абстрагироваться от такой сложности, мы будем рассматривать DFF как примитивный строительный блок. В частности, симулятор аппаратуры «От Nand до “Тетриса”» имеет встроенную реализацию DFF, которую легко использовать для создания других микросхем, которые мы сейчас и рассмотрим.

3.4.2. Регистры

Микросхемы регистров — это устройства памяти: предполагается, что они реализуют базовое поведение $out(t + 1) = out(t)$, запоминая и подавая на выход свое состояние с течением времени. Их поведение похоже на поведение DFF: $out(t + 1) = in(t)$. Если подать выход DFF обратно на его вход, то это станет хорошей отправной точкой для реализации однобитового регистра. Данное решение показано слева на иллюстрации 3.9.

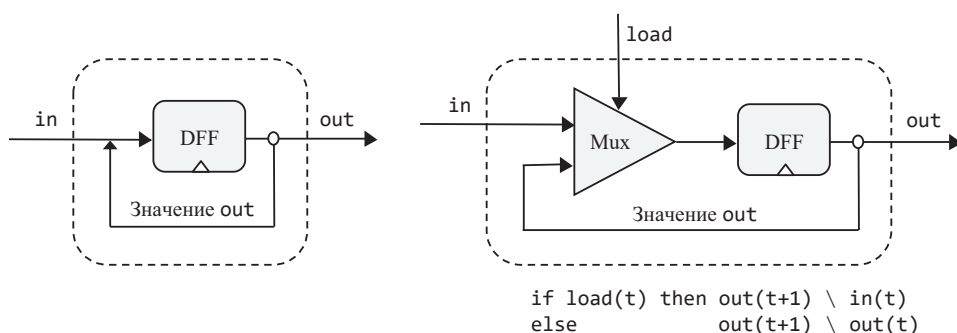


Иллюстрация 3.9. Реализация однобитного реестра *Bit*: неверное (слева) и верное (справа) решения.

Но само по себе такое решение, показанное в левой части иллюстрации 3.9, неверно по нескольким взаимосвязанным причинам. Во-первых, реализация не подразумевает наличия бита *load*, что требуется по интерфейсу. Во-вторых, нет никакого способа указать DFF, когда он должен брать входные данные из входа *in*, а когда из выхода *out*. И в самом деле, правила программирования HDL запрещают

подавать на один контакт входа контакты вывода более чем от одного источника.

Впрочем, неверная реализация подталкивает нас к верной, показанной в правой части иллюстрации 3.9. Как следует из схемы чипа, единственный способ разрешить неоднозначность входа — ввести в конструкцию мультиплексор. Бит загрузки `load` общей микросхемы можно направить на бит выбора внутреннего мультиплексора: если мы зададим этому биту значение 1, мультиплексор будет подавать на DFF значение `in`; если задать биту `load` значение 0, мультиплексор подаст на вход DFF прежнее значение выхода. Так реализуется поведение «если `load`, записать в регистр новое значение, иначе сохранить прежнее значение», а именно это мы и хотим от регистра.

Обратите внимание, что только что описанный цикл обратной связи не влечет за собой проблем в виде так называемой *гонки данных* (состызания между сигналами, состояния неоднозначности при передаче сигналов со временем): цикл проходит через вентиль DFF, реализующий временную задержку. По своей сути показанная на иллюстрации 3.9 конструкция `Bit` является частным случаем общей конструкции секвенциальной логики, изображенной на иллюстрации 3.4.

Завершив реализацию однобитного регистра `Bit`, мы можем перейти к построению n -битного регистра. Сделать это можно посредством формирования массива из n чипов `Bit` (см. рисунок 3.1). Основным параметром конструкции такого регистра — это n , то есть количество бит, которое он должен хранить, например, 16, 32 или 64. Поскольку компьютер Hack будет основан на 16-битной аппаратной платформе, наша микросхема `Register` будет основана на шестнадцати однобитовых микросхемах-частях `Bit`.

Регистр `Bit` — единственная микросхема в архитектуре Hack, использующая вентиль DFF напрямую; все устройства памяти более высокого уровня в компьютере используют вентили DFF косвенно, поскольку включают в свой состав микросхемы `Register`, состоящие из микросхем `Bit`. Обратите внимание, что использование вентиля DFF в конструкции любой микросхемы — прямое или косвенное — превращает данную микросхему, а также все микросхемы более высокого

уровня, использующие ее в качестве составной части, в последовательностные микросхемы, зависящие от времени.

3.4.3. ОЗУ

Для аппаратной платформы Hack потребуется ОЗУ из 16К (16384) 16-битовых регистров, так что его мы и будем реализовывать. Итак, предлагаем следующий поэтапный план построения.

Микросхема	n	k	Создается из:
RAM8	8	3	8 микросхем Register
RAM64	64	6	8 микросхем RAM8
RAM512	512	9	8 микросхем RAM64
RAM4K	4096	12	8 микросхем RAM512
RAM16K	16384	14	4 микросхемы RAM4K

Все эти микросхемы памяти имеют точно такой же API RAM_n , как показано на рисунке 3.7. Каждая микросхема ОЗУ (RAM) имеет n регистров, а ширина ее адресного входа составляет $k = \log_2 n$ бит. Теперь опишем, как можно реализовать данные микросхемы, начиная с RAM8.

Микросхема RAM8 имеет 8 регистров, как показано на иллюстрации 3.7, для $n = 8$. Каждый регистр можно выбрать посредством подачи на 3-битный адресный вход RAM8 значения от 0 до 7. Чтение значения выбранного регистра можно описать следующим образом: как при наличии адреса `address` (значения между 0 и 7) «выбрать» регистр с номером `address` и направить его значение на выход RAM8? *Подсказка:* это можно сделать с помощью одной из комбинационных микросхем, построенных в проекте 1. Поэтому чтение значения выбранного регистра ОЗУ происходит практически мгновенно, независимо от тактового генератора и количества регистров в ОЗУ. Аналогично акт записи значения в выбранный регистр можно описать следующим образом: как при наличии значения `address`, значения `load` (1) и 16-битного значения `in` записать в регистр с адресом `address` значение `in`? *Подсказка:* 16-битное значение `in` можно подавать одновременно на входы `in` всех восьми микросхем регистра. С помощью

другой разработанной в проекте 1 комбинационной микросхемы можно добиться того, что только один из регистров примет входящее значение `in`, а все остальные семь регистров проигнорируют его.

Заметим мимоходом, что регистры оперативной памяти не обозначены адресами в каком-либо физическом смысле. Описанной выше логики достаточно для выбора отдельных регистров в соответствии с их адресом, и это делается благодаря использованию комбинационных микросхем. Теперь следует сделать чрезвычайно важное замечание: поскольку комбинационная логика не зависит от времени, время доступа к любому отдельному регистру будет почти мгновенным.

После реализации микросхемы RAM8 перейдем к реализации микросхемы RAM64. Ее реализация может быть основана на восьми микросхемах RAM8. Для выбора конкретного регистра из памяти RAM64 используется 6-битный адрес, допустим, `xxxuuu`. Биты `xxx` используются для выбора одной из микросхем RAM8, а биты `uuu` используются для выбора одного из регистров в выбранной RAM8. Такую иерархическую схему адресации можно реализовать с помощью логики вентиля. Ту же идею можно положить в основу реализации остальных микросхем RAM512, RAM4K и RAM16K.

Вкратце мы берем совокупность неограниченного числа регистров и накладываем на нее комбинационную надстройку, позволяющую получать прямой доступ к любому отдельному регистру. Надеемся, что красота этого решения не ускользнет от внимания читателя.

3.4.4. Счетчик

Счетчик — это устройство памяти, способное увеличивать свое значение в каждую единицу времени. Кроме того, значение счетчика можно установить на 0 или любое другое. Основные функции хранения и вычисления значения счетчика можно реализовать соответственно микросхемой регистра и микросхемой инкрементора, построенной в проекте 2. Логiku выбора между режимами работы счетчика `inc`, `load` и `reset` можно реализовать с помощью некоторых мультиплексоров, построенных в проекте 1.

3.5. Проект

Задача: построить все микросхемы, описанные в данной главе. В качестве строительных блоков можно использовать примитивные вентили DFF, построенные на их основе микросхемы и вентили, построенные в предыдущих главах.

Ресурсы: единственный инструмент, который вам понадобится для этого проекта, — симулятор аппаратуры «От Nand до “Тетриса”». Все микросхемы должны быть реализованы на языке HDL, указанном в приложении 2. Как обычно, для каждой микросхемы мы предоставляем каркасную программу `.hdl` с недостающей частью реализации, файл сценария `.tst`, указывающий симулятору аппаратуры, как ее тестировать, и файл сравнения `.cmp`, определяющий ожидаемые результаты. Ваша задача — дописать недостающие части реализации в предоставленных программах `.hdl`.

Контракт: при загрузке в симулятор аппаратуры ваш проект микросхемы (модифицированная программа `.hdl`), протестированный на прилагаемом файле `.tst`, должен подавать на выходы значения, перечисленные в прилагаемом файле `.cmp`. Если это не так, симулятор сообщит вам об этом.

Совет: вентиль триггера данных (DFF) считается примитивным, поэтому его создавать не нужно. Встречая в HDL-программе часть в виде DFF, симулятор автоматически вызывает реализацию `tools/builtIn/DFF.hdl`.

Структура папок данного проекта: при конструировании микросхем ОЗУ из микросхем-частей ОЗУ более низкого уровня рекомендуется использовать встроенные версии последних. В противном случае симулятор будет рекурсивно генерировать множество занимающих память программных объектов, по одному для каждой из многочисленных микросхем-частей, составляющих типичный

блок ОЗУ. Это может привести к медленной работе симулятора или, что еще хуже, к исчерпанию памяти компьютера, на котором запущен симулятор.

Чтобы не допустить такой проблемы, мы разделили файлы создаваемых в этом проекте микросхем ОЗУ на две подпапки. Программы `RAM8.hdl` и `RAM64.hdl` хранятся в `projects/03/a`, а другие, более высокоуровневые микросхемы RAM — в `projects/03/b`. Такое разделение сделано только с одной целью: при оценке микросхем RAM, хранящихся в папке `b`, симулятор будет вынужден использовать встроенные реализации частей микросхемы RAM64, поскольку в папке `b` он не сможет найти файл `RAM64.hdl`.

Шаги: мы рекомендуем действовать в следующем порядке.

1. Симулятор аппаратуры, необходимый для данного проекта, расположен в папке `nand2tetris/tools`.
2. При необходимости обратитесь к приложению 2 и к руководству по использованию симулятора аппаратуры.
3. Соберите и смоделируйте все микросхемы, указанные в папке `projects/03`.

Веб-версия проекта 3 доступна на сайте www.nand2tetris.org.

3.6. Перспектива

Краеугольным камнем всех систем памяти, описанных в этой главе, является триггер, который мы рассматривали абстрактно, как примитивный встроенный вентиль. Типичный подход заключается в построении триггеров из элементарных комбинационных вентилей (например, Nand), соединяемых в контуры обратной связи. Стандартная конструкция начинается с построения асинхронного (не связанного с генератором тактовой частоты) бистабильного триггера, то есть триггера, способного принимать одно из двух устойчивых состояний (хранить 0 или 1). Затем посредством каскадного соединения двух

таких асинхронных триггеров получается синхронный триггер; первая его часть устанавливается при повышении уровня сигнала синхронизации («тик»), а вторая при его понижении («так»). Такая схема дизайна «ведущий — ведомый» наделяет триггер требуемой функциональностью синхронизации с генератором тактовой частоты.

Такие реализации триггеров одновременно элегантны и сложны. В данной книге мы решили абстрагироваться от этих низкоуровневых схем и рассматривать триггер как примитивный вентиль. Читатели, желающие изучить внутреннюю структуру триггеров, могут найти подробные описания в большинстве учебников по проектированию цифровых устройств и компьютерной архитектуре.

Одна из причин не останавливаться на подробной схеме триггеров заключается в том, что самый низкий уровень устройств памяти, используемых в современных компьютерах, не обязательно строится на триггерах. Вместо этого современные микросхемы памяти тщательно оптимизируются на основе уникальных физических свойств материалов и устройств, используемых для хранения данных. Сегодня разработчикам компьютеров доступно множество таких альтернативных технологий; как обычно, вопрос о том, какую технологию использовать, зависит от соотношения цены и качества. Точно так же довольно элегантен, но не обязательно эффективен метод рекурсивного восхождения, который мы использовали для создания чипов оперативной памяти. Возможны и более эффективные реализации.

Если же отвлечься от этих физических соображений, то все описанные в данной главе конструкции микросхем — регистры, счетчик и микросхемы оперативной памяти — являются стандартными, и их версии можно найти в любой компьютерной системе.

В главе 5 мы будем использовать микросхемы регистров, созданные в данной главе, вместе с АЛУ, созданным в главе 2, для построения центрального процессора. Затем центральный процессор будет дополнен устройством оперативной памяти, что приведет к созданию архитектуры компьютера общего назначения, способного выполнять программы, написанные на машинном языке. Этот машинный язык рассматривается в следующей главе.

4. Машинный язык

Произведения на основе вымысла следует писать очень простым языком; чем больше в них чистого воображения, тем сильнее требование простоты.

— Сэмюэл Тейлор Кольридж (1772–1834)

В главах 1–3 мы построили микросхемы обработки данных и микросхемы памяти, которые можно будет интегрировать в аппаратную платформу универсального компьютера. Но прежде чем приступить к завершению данной конструкции, сделаем паузу и спросим себя: каково *назначение* этого компьютера? Согласно знаменитому высказыванию архитектора Луиса Салливана, «форма следует за функцией». Иными словами, если вы хотите понять систему или построить ее, начните с изучения функции, которую эта система должна выполнять. Исходя из этого, до того, как приступить к созданию нашей аппаратной платформы, мы решили посвятить текущую главу изучению *машинного языка*, для реализации которого эта платформа предназначена. В конце концов, конечная функция любого универсального компьютера, или компьютера общего назначения — это эффективное выполнение программ, написанных на машинном языке.

Машинный язык — совокупность заранее установленных правил, предназначенных для кодирования машинных команд, или инструкций. С помощью команд мы поручаем процессору компьютера выполнять арифметические и логические операции, читать значения из памяти компьютера и записывать значения в его память, проверять

булевы условия и решать, какую команду следует извлечь и выполнить следующей. В отличие от языков высокого уровня, цель разработки которых заключается в обеспечении кроссплатформенной совместимости и ясного способа выражения сложных команд, машинные языки предназначены для непосредственного исполнения операций на конкретной аппаратной платформе при полном контроле ее возможностей. Конечно, общий характер языка, его элегантность и понятность остаются желанными качествами, но только в той степени, в которой они поддерживают основное требование прямого и эффективного выполнения операций на основе конкретной аппаратной платформы.

Машинный язык — это самый глубокий интерфейс взаимодействия человека и компьютера, та тонкая грань, где аппаратное обеспечение встречается с программным. Это точка, где абстрактные замыслы людей, выраженные в программах высокого уровня, в конечном счете сводятся к физическим операциям, выполняемым микросхемами. Таким образом, машинный язык можно рассматривать и как часть сферы программирования, и как неотъемлемую часть аппаратной платформы. Фактически точно так же как мы говорим, что машинный язык предназначен для управления определенной аппаратной платформой, можно сказать, что аппаратная платформа предназначена для выполнения инструкций, написанных на определенном машинном языке.

Глава начинается с общего введения в низкоуровневое программирование на машинном языке. Далее мы приводим подробную спецификацию *машинного языка Hack*, охватывающую как его двоичную, так и символьную версии. Проект, которым заканчивается глава, посвящен написанию нескольких программ на машинном языке. Он позволит читателям получить практическое представление о низкоуровневом программировании и заложит основу для завершения создания аппаратного обеспечения компьютера в следующей главе.

Хотя программисты редко пишут программы непосредственно на машинном языке, изучение низкоуровневого программирования — необходимое условие для полного и глубокого понимания того, как работают компьютеры. Кроме того, глубокое понимание низкоуровневого программирования помогает программисту писать более

качественные и эффективные программы высокого уровня. Наконец, довольно увлекательно наблюдать на практике за тем, как самые сложные программные системы оказываются, по сути, потомками простых инструкций, каждая из которых задумана выполнять побитовую операцию на аппаратном уровне.

4.1. Машинный язык: обзор

Эта глава посвящена не столько слову «машинный», сколько слову «язык», с помощью которого контролируют машину. Поэтому мы абстрагируемся от аппаратной платформы и сосредотачиваемся только на том минимальном подмножестве элементов аппаратуры, которые явно упоминаются в командах машинного языка.

4.1.1. Аппаратные элементы

Машинный язык можно рассматривать как формальное средство, набор заранее определенных команд для манипулирования *памятью* с помощью *процессора* и набора *регистров*.

Память: термин «*память*» в широком смысле охватывает множество аппаратных устройств, хранящих данные и команды в компьютере. С функциональной точки зрения память представляет собой непрерывную последовательность ячеек, также называемых *участками* или *регистрами памяти*, каждая из которых имеет уникальный *адрес*. Доступ к отдельному регистру памяти осуществляется посредством ввода его адреса.

Процессор: процессор, обычно называемый *центральным процессором* или *центральным процессорным устройством* (ЦП, ЦПУ), — это устройство, способное выполнять фиксированный набор примитивных операций. К ним относятся арифметические и логические операции, операции доступа к памяти и операции управления (также называемые *ветвлением*). Процессор получает входные данные

из выбранных регистров и участков памяти и записывает выходные данные в выбранные регистры и участки памяти. Он состоит из АЛУ, множества регистров и логических вентилей, позволяющих ему анализировать и выполнять команды в виде двоичного кода.

Регистры: процессор и память реализованы как две отдельные, самостоятельные микросхемы, и перемещение данных из одной в другую происходит относительно медленно. По этой причине процессоры обычно оснащаются несколькими встроенными регистрами, каждый из которых способен хранить одно значение. Эти расположенные внутри микросхемы процессора регистры служат высокоскоростной локальной памятью, позволяющей процессору манипулировать данными и командами без необходимости выходить за пределы микросхемы.

Регистры внутри ЦП делятся на две категории: *регистры данных*, предназначенные для хранения значений данных, и *адресные регистры* (*индекс-регистры*), предназначенные для хранения значений, которые могут быть интерпретированы либо как значения данных, либо как адреса памяти. Архитектура компьютера устроена таким образом, что помещение определенного значения, скажем, n , в адресный регистр приводит к тому, что мгновенно* *выбирается* участок памяти, адрес которого равен n . Это позволяет выполнить последующую операцию над выбранным участком памяти.

4.1.2. Языки

Программы на машинном языке можно записывать двумя альтернативными, но эквивалентными способами: *двоичным* и *символьным*. Рассмотрим, например, абстрактную операцию «записать в регистр R1 значение $R1 + R2$ ». Допустим, мы, как разработчики языка, решили передавать операцию сложения 6-битным кодом 101011, а регистры R1 и R2 — кодами 00001 и 00010 соответственно. Записав данные коды слева направо, получаем 16-битную команду 1010110001000001

* Под словом «мгновенно» мы имеем в виду «в тот же тактовый цикл, или единицу времени».

в качестве двоичного варианта операции «записать в регистр R1 значение $R1 + R2$ ».

На заре развития компьютерных систем вычислительные машины программировались вручную: когда первые протопрограммисты хотели подать машине команду «записать в регистр R1 значение $R1 + R2$ », они устанавливали в разные положения механические переключатели, подающие сигналы и записывающие двоичный код типа 1010110001000001 в память инструкций вычислительной машины. Если программа состояла из ста команд, им приходилось проходить через это испытание сто раз. Конечно, отладка таких программ превращалась в настоящий кошмар. Это заставило программистов придумать и использовать символьные коды как удобный способ документирования и отладки программ *на бумаге*, прежде чем вводить их в компьютер. Например, команду «записать в регистр R1 значение $R1 + R2$ » можно представить в символьском формате как «add R2, R1», и эта запись будет соответствовать команде 1010110001000001 в двоичном формате*.

Чуть погодя некоторые специалисты пришли к другой подобной идее: такие символы, как R, 1, 2 и +, тоже можно представлять с помощью установленных заранее двоичных кодов. Почему бы не использовать для написания программ символьные команды, а затем не воспользоваться другой программой — *транслятором* («переводчиком») — для перевода символьных команд в исполняемый двоичный код? Эта инновация освободила программистов от утомительной работы по написанию двоичного кода и открыла путь для последующей разработки высокоуровневых языков программирования.

По причинам, которые станут понятны в главе 6, символьные машинные языки называются *языками ассемблера*, или *ассемблерными языками* («языками сборки»), а программы, которые переводят их в двоичный код, называются *ассемблерами* («сборщиками»).

В отличие от синтаксиса языков высокого уровня, допускающего использование на разных компьютерах, синтаксис ассемблерного

* Здесь и до конца раздела 4.1 приводятся примеры команд на условном «типичном языке ассемблера», а не на конкретном машинном языке Наск, который будет описан далее. — *Прим. пер.*

языка тесно связан с низкоуровневыми особенностями целевого оборудования: доступными операциями АЛУ, количеством и типом регистров, размерами памяти и т. д.

Поскольку разные компьютеры сильно различаются по любому из этих параметров, то существует огромное количество (целое вавилонское столпотворение) машинных языков, каждый со своим неочевидным синтаксисом, и каждый разработан для управления определенных семейств процессоров. Однако, независимо от этого разнообразия, все машинные языки теоретически эквивалентны, и все они поддерживают схожие наборы общих задач, которые мы сейчас и опишем.

4.1.3. Команды

В дальнейшем мы будем считать, что процессор компьютера оснащен множеством регистров, обозначаемых R0, R1, R2... Точное количество и типы этих регистров не имеют значения для нашего обсуждения.

Арифметические и логические операции: каждый машинный язык имеет команды для выполнения основных арифметических операций, таких как сложение и вычитание, а также основных логических операций, таких как And, Or, Not. Рассмотрим, например, следующие сегменты кода.

```
// Сложение двух чисел:
```

```
load R1, 17           // R1 ← 17
```

```
load R2, 4            // R2 ← 4
```

```
add R1, R1, R2        // R1 ← R1 + R2
```

```
// Вычисление логической операции:
```

```
load R1, true         // R1 ← двоичное представление true
```

```
load R2, false        // R2 ← двоичное представление false
```

```
and R1, R1, R2         // R1 ← R1 And R2 (побитовое And)
```

Чтобы такие символьные команды выполнялись на компьютере, их сначала нужно перевести в двоичный код. Перевод выполняется

программой, называемой *ассемблером*, которую мы рассмотрим в главе 6. Пока что мы предполагаем, что у нас есть доступ к такому ассемблеру и что мы можем использовать его по мере необходимости.

Доступ к памяти: каждый машинный язык имеет средства для доступа к выбранным ячейкам памяти и последующего манипулирования ими. Обычно для этого используется *адресный регистр*, назовем его *A*. Предположим, мы хотим записать в ячейку памяти номер 17 значение 1. Сделать это можно с помощью двух команд: `load A, 17` и затем `load M, 1`, где по соглашению *M* означает регистр памяти, выбранный *A* (а именно регистр памяти, адрес которого является текущим значением регистра *A*). Теперь предположим, что мы хотим записать значение 1 в пятьдесят ячеек памяти 200, 201, 202, ..., 249. Это можно сделать, отдав команду `load A, 200`, а затем запустив цикл, который пятьдесят раз выполнит команды `load M, 1` и `add A, A, 1`.

С использованием
физических адресов

```
...
// Sets R1 to 0+1+2, ...
12: load R1,0
13: add R1,R1,1
...
27: goto 13
...
```

С использованием
символических адресов

```
...
// Sets R1 to 0+1+2, ...
    load R1,0
(L0OP)
    add R1,R1,1
    ...
    goto L0OP
...
```

Иллюстрация 4.1. Две версии одного и того же низкоуровневого кода (предполагается, что код включает в себя не показанную здесь логику выхода из цикла).

Контроль потока (управление потоком): хотя компьютерные программы по умолчанию выполняются последовательно, одна команда за другой, в них могут фигурировать *переходы* (англ. «*jumps*» — «прыжки») к командам, идущим не за текущей, а расположенным где-то в другом месте программы. Чтобы облегчить такое ветвление, машинные языки имеют несколько вариантов условных и безусловных

команд *goto* («перейти на»), а также средства объявления меток, задающие цель для перехода *goto*. На иллюстрации 4.1 показано простое ветвление с помощью машинного языка.

Символы: обе версии кода на иллюстрации 4.1 написаны на языке ассемблера, поэтому перед выполнением их необходимо перевести в двоичный код. Кроме того, обе версии выполняют совершенно одинаковую логику. Однако версия кода, использующая символические ссылки, намного проще в написании, отладке и работе.

Кроме того, в отличие от кода с физическими адресами, переведенную двоичную версию кода с символическими ссылками можно загрузить в любой сегмент памяти, который окажется доступным в памяти компьютера, и выполнить из него. Поэтому низкоуровневый код, где не упоминаются физические адреса, считается *перемещаемым*. Очевидно, что перемещаемый код просто необходим в таких компьютерных системах, как ПК и мобильные телефоны, регулярно загружающих и выполняющих множество приложений динамически и одновременно. Таким образом, мы видим, что символические ссылки — это не просто косметические функции, они используются для освобождения кода от ненужных физических привязок к памяти компьютера.

На этом мы заканчиваем краткое знакомство с некоторыми основными элементами машинных языков. В следующем разделе дается формальное описание одного конкретного машинного языка — «родного» (нативного) кода компьютера Hack.

4.2. Машинный язык Hack

Программисты, составляющие низкоуровневый код (или компиляторы и интерпретаторы, генерирующие низкоуровневый код), взаимодействуют с компьютером абстрактно, через его интерфейс, которым служит машинный язык компьютера. Хотя программисты и не обязаны знать все особенности архитектуры компьютера, они должны быть знакомы с элементами аппаратного обеспечения, которые задействуются в их низкоуровневых программах.

Учитывая это, начнем обсуждение машинного языка Наск с концептуального описания компьютера Наск. Далее мы приведем пример полной программы, написанной на языке ассемблера Наск. Это создаст основу для оставшейся части данного раздела, в которой мы дадим формальную спецификацию команд языка Наск.

4.2.1. Основы языка Наск

Конструкция компьютера Наск, которая будет описана в следующей главе, соответствует широко используемой аппаратной парадигме «*архитектура фон Неймана*», названной в честь пионера вычислительной техники Джона фон Неймана. Наск — это 16-битный компьютер (то есть центральный процессор и блоки памяти предназначены для обработки, передачи и хранения 16-битных значений).

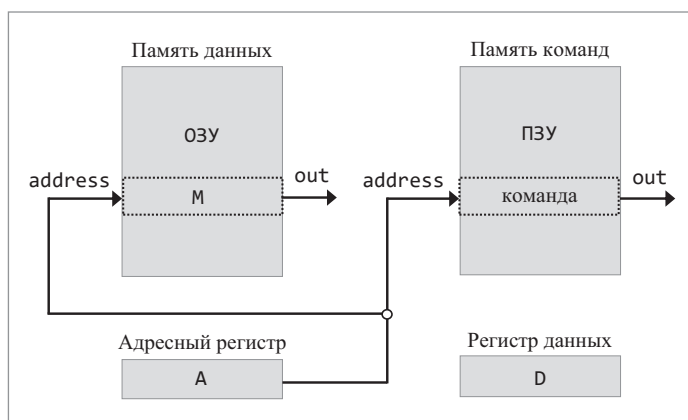


Иллюстрация 4.2. Концептуальная модель системы памяти Наск. Несмотря на то что реальная архитектура построена несколько иначе (как описано в главе 5), эта модель помогает понять семантику программ Наск.

Память: как показано на иллюстрации 4.2, платформа Наск использует два отдельных блока памяти: память данных и память команд. В памяти данных хранятся двоичные значения, которыми манипулируют программы. В памяти команд хранятся команды программы, также представленные в виде двоичных значений. Обе памяти

имеют ширину 16 бит, и каждая из них имеет 15-битное адресное пространство. Таким образом, максимальный адресуемый размер каждого блока памяти составляет 2^{15} или 32К 16-битных слов (символ К, сокращение от *kilo*, греческого слова, обозначающего тысячу, обычно используется для обозначения числа $2^{10} = 1024$). Удобно рассматривать каждый блок памяти как линейную последовательность адресуемых регистров памяти с адресами от 0 до 32К-1.

Память данных (которую мы также называем ОЗУ) — это устройство чтения/записи. Команды Наск могут считывать данные из выбранных регистров ОЗУ и записывать данные в них. Выбор отдельного регистра осуществляется посредством ввода его адреса. Поскольку вход *address* памяти всегда содержит некоторое значение, всегда будет иметься некий выбранный регистр, и этот регистр в командах Наск обозначается как М. Например, Наск-команда $M = 0$ устанавливает в выбранный регистр ОЗУ значение 0.

Память команд (или *память инструкций*, которую мы также называем ПЗУ) — это устройство, предназначенное только для чтения, и программы загружаются в нее с помощью некоторых внешних средств (подробнее об этом в главе 5). Как и в случае с ОЗУ, вход *address* памяти команд всегда содержит некоторое значение; поэтому один из регистров памяти команд всегда будет выбранным. Значение этого регистра считается *текущей командой*.

Регистры: команды Наск предназначены для работы с тремя 16-битными регистрами: *регистром данных*, обозначаемым D, *адресным регистром*, обозначаемым A, и выбранным регистром памяти данных, обозначаемым M. Команды Наск имеют понятный синтаксис. Например: $D = M$, $M = D + 1$, $D = 0$, $D = M - 1$ и т. д.

Роль регистра данных D проста: он служит для хранения некоего 16-битного значения. Второй регистр, A, служит одновременно регистром адреса и регистром данных. Если мы хотим сохранить значение 17 в регистре A, то используем Наск-команду @17 (причина такого синтаксиса скоро станет понятна). Фактически это единственный способ ввести в компьютер Наск некую константу. Например, если мы хотим записать значение 17 в регистр D, то нужно будет использовать две команды:

@17, а затем $D = A$. Помимо того, что он служит вторым регистром данных, «трудолюбивый» регистр A также используется для адресации памяти данных и адресации памяти команд, о чем мы сейчас и поговорим.

Адресация: Hack-команда @xxx записывает в регистр A значение xxx. Кроме того, команда @xxx имеет два побочных эффекта. Во-первых, она делает регистр ОЗУ с адресом xxx выбранным участком памяти, обозначаемым M . Во-вторых, делает значение регистра ПЗУ с адресом xxx выбранной командой. Поэтому установка значения в регистр A в какой-то степени подготавливает почву сразу для одного из двух очень разных последующих действий: манипулирования выбранным участком памяти данных или выполнения чего-то с выбранной командой. Какое действие следует выполнить (а какое проигнорировать), определяется последующей Hack-командой.

Для примера предположим, что мы хотим записать значение 17 в ячейку ОЗУ $RAM[100]$. Это можно сделать с помощью следующих Hack-команд: @17, $D = A$, @100, $M = D$. Обратите внимание, что в первой паре команд A служит регистром данных, а во второй — адресным регистром. Приведем другой пример: чтобы записать в ячейку $RAM[100]$ значение из ячейки $RAM[200]$, можно воспользоваться следующими Hack-командами: @200, $D = M$, @100, $M = D$.

В обоих этих сценариях регистр A также выбирал регистры и в памяти команд, то есть совершал действие, которое в этих двух сценариях игнорируется. В следующем разделе рассматривается противоположный сценарий: использование регистра A для выбора команд при игнорировании его влияния на память данных.

Ветвление: приведенные выше примеры кода подразумевают, что программа Hack — это последовательность команд, которые должны выполняться одна за другой. И действительно, это стандартный порядок выполнения программы, но что, если мы захотим перейти не к выполнению следующей команды, а, скажем, к команде номер 29 в списке инструкций программы? В языке Hack это можно сделать с помощью команды @29, за которой следует команда 0; JMP. Первая команда выбирает регистр ПЗУ $ROM[29]$ (заодно выбирая и $RAM[29]$, но нас

это сейчас не заботит). Вторая команда 0; JMP реализует операцию *безусловного ветвления*, а именно переход к выполнению инструкции, адресуемой регистром А (префикс 0; мы объясним позже). Поскольку предполагается, что ПЗУ (ROM) уже содержит нашу программу, выполняемую с адреса 0, то две команды @29 и 0; JMP в итоге делают значение ячейки ROM[29] следующей выполняемой командой.

В языке Nask также предусмотрена возможность *условного ветвления*. Например, логика if D == 0 goto 52 (если D == 0, перейти к 52) может быть реализована с помощью команды @52, за которой следует команда D; JEQ. Семантика второй команды такова: «оценить D; если значение равно нулю, перейти к выполнению команды, хранящейся по адресу, выбранному А». Как мы объясним далее в этой главе, в языке Nask имеется несколько таких команд условного ветвления.

Итак, говоря вкратце: регистр А выполняет две одновременные, но очень разные функции адресации. После команды @xxx мы либо имеем дело с выбранным регистром памяти данных (М) и игнорируем выбранную команду, либо имеем дело с выбранной командой и игнорируем выбранный регистр памяти данных. Эта двойственность немного сбивает с толку, но обратите внимание, что нам удалось применить один адресный регистр для управления двумя отдельными устройствами памяти (см. иллюстрацию 4.2). В результате мы получили более простую архитектуру компьютера и компактный машинный язык. Как обычно, главные принципы в нашем деле — простота и экономность.

Переменные: xxx в Nask-команде @xxx может быть либо константой, либо символом. Если выполняется команда @23, в регистр А записывается значение 23. Если выполняется команда @x, где x — символ, связанный с некоторым значением, например, 513, то команда записывает в регистр А значение 513. Язык ассемблера Nask позволяет использовать *переменные*, а не физические адреса памяти. Например, типичный оператор присваивания в языке высокого уровня let x = 17 можно реализовать на языке Nask следующим образом: @17, D = A, @x, M = D. Семантика данного кода такова: «выбрать регистр ОЗУ (RAM), адрес которого является значением, связанным с символом x, и установить в этом регистре значение 17». Здесь мы предполагаем, что существует

некое средство, умеющее связывать символы, встречающиеся в языках высокого уровня (такие как *x*), с разумными и последовательными адресами в памяти данных. И это средство — ассемблер.

Примеры доступа к памяти	Примеры ветвления	Примеры использования переменных
<pre>// D \ 17 @17 D\A // RAM[100] \ 17 @17 D\A @100 M\D // RAM[100] \ RAM[200] @200 D\M @100 M\D</pre>	<pre>// goto 29 @29 0;JMP // if D>0 goto 63 @63 D;JGT</pre>	<pre>// x \ -1 @x M\ -1 // count \ count - 1 @count M\M-1 // sum \ sum + x @sum D\M @x D\D+M @sum M\D</pre>

Иллюстрация 4.3. Примеры кода языка ассемблера Nask.

Благодаря ассемблеру в программах Nask можно по желанию и по мере необходимости именовать и использовать переменные. Предположим, например, что мы хотим написать код, реализующий счетчик. Один из вариантов — хранить данные этого счетчика, скажем, в ячейке RAM[30] и увеличивать его значение с помощью команд @30, M = M + 1. Однако более разумным вариантом будет набор команд @count, M = M + 1, и пусть ассемблер сам беспокоится о том, в какую ячейку памяти из доступных поместить эту переменную. Нам не важен конкретный адрес — лишь бы ассемблер всегда сопоставлял символ с этим адресом. В главе 6 мы узнаем, как разработать ассемблер, реализующий эту полезную операцию сопоставления.

В дополнение к символам, которые можно вводить в ассемблерные программы Nask по мере необходимости, язык Nask имеет шестнадцать встроенных символов R0, R1, R2, ..., R15. Эти символы всегда привязываются ассемблером к значениям 0, 1, 2, ..., 15. Так,

например, две Наск-команды $@R3, M = 0$ в итоге записывают в ячейку $RAM[3]$ значение 0. В дальнейшем мы иногда будем называть $R0, R1, R2, \dots, R15$ *виртуальными регистрами*.

Прежде чем продолжить, мы предлагаем вам просмотреть примеры кода на иллюстрации 4.3 и убедиться, что вы полностью поняли их (некоторые из них уже обсуждались).

4.2.2. Пример программы

Давайте для начала, фигурально выражаясь, сразу прыгнем в воду и рассмотрим полную программу ассемблерного языка Наск, отложив его формальное описание до следующего раздела. Но прежде несколько слов предостережения: большинство читателей, вероятно, будут озадачены непонятным стилем данной программы. На что мы отвечаем: добро пожаловать в программирование на машинном языке! В отличие от языков высокого уровня, машинные языки не предназначены для того, чтобы радовать программистов. Они скорее предназначены для эффективного и четкого управления аппаратной платформой.

Предположим, мы хотим вычислить сумму $1 + 2 + 3 + \dots + n$ для заданного значения n . В ходе выполнения операции поместим входное значение n в регистр $RAM[0]$, а выходную сумму в $RAM[1]$. Программа, которая вычисляет эту сумму, показана на иллюстрации 4.4. В левой ее части показан псевдокод (*прим. переводчика*: представление на условном «языке высокого уровня»), дающий более понятное представление об алгоритме. Обратите внимание, что вместо использования известной формулы для вычисления суммы арифметического ряда мы используем «грубое» сложение. Это сделано для иллюстрации условной и итеративной обработки на машинном языке Наск.

Продолжив читать главу, вы поймете данную программу полностью. Пока же мы предлагаем игнорировать детали, а вместо этого обратить внимание на следующую закономерность: в языке Наск каждая операция с ячейкой памяти состоит из двух команд. Первая команда $@addr$ используется для выбора целевого адреса памяти; последующая команда определяет, что делать по этому адресу. В языке Наск такую логику поддерживают две общие команды, примеры которых мы

уже видели: *адресная команда*, также называемая *А-командой* (команды, начинающиеся со знака @), и *вычислительная команда*, также называемая *С-командой* (от англ. слова «*computation*» — «вычисление», все остальные команды). Каждая команда имеет символьное и двоичное представление; эффект же их мы опишем ниже.

Псевдокод

```
// Program: Sum1ToN
// Computes RAM[1]\1+2+3+...+RAM[0]
// Usage: put a value>\1 in RAM[0]
i \ 1
sum \ 0
LOOP:
  if (i > R0) goto STOP
  sum \ sum + i
  i \ i + 1
  goto LOOP
STOP:
  R1 \ sum
```

Код языка ассемблера Hack

```
// File: Sum1ToN.asm
// Computes RAM[1]\1+2+3+...+RAM[0]
// Usage: put a value>\1 in RAM[0]
// i \ 1
@i
M\1
// sum \ 0
@sum
M\0
(LLOOP)
// if (i > R0) goto STOP
@i
D\M
@R0
D\D-M
@STOP
D;JGT
// sum \ sum + i
@sum
D\M
@i
D\D+M
@sum
M\D
// i \ i + 1
@i
M\M+1
// goto LOOP
@LLOOP
0;JMP
(STOP)
// R1 \ sum
@sum
D\M
@R1
M\D
(END)
@END
0;JMP
```

Иллюстрация 4.4. Пример программы на языке ассемблера Hack. Обратите внимание, что на RAM[0] и RAM[1] можно ссылаться как R0 и R1.

4.2.3. Спецификация языка Nask

В машинном языке Nask имеются два типа команд, показанных на иллюстрации 4.5.

Символическая: @xxx
A-команда:
Двоичная: 0 vvvvvvvvvvvvvvvv

Символическая: dest \ comp; jump
C-команда:
Двоичная: 111 a c c c c c c d d d j j j

(xxx — двоичное значение от 0 до 32 767 или символ, привязанный к такому двоичному значению)
(vv ... v = 15-битное значение xxx)
(поле comp — обязательно; если поле dest пусто, = опускается; если поле jump пусто, ; опускается)

comp					dest				Эффект: результат comp сохраняется в:				
c	s	s	s	s	d	d	d	d					
0			1	0	1	0	1	0	null	0	0	0	значение не сохраняется RAM[A] D register (reg) D reg и RAM[A] A reg A reg и RAM[A] A reg и D reg A reg, D reg, и RAM[A]
1			1	1	1	1	1	1	M	0	0	1	
-1			1	1	1	0	1	0	D	0	1	0	
D			0	0	1	1	0	0	DM	0	1	1	
A	M		1	1	0	0	0	0	A	1	0	0	
!D			0	0	1	1	0	1	AM	1	0	1	
!A	!M		1	1	0	0	0	1	AD	1	1	0	
-D			0	0	1	1	1	1	ADM	1	1	1	
-A	-M		1	1	0	0	1	1					
D+1			0	1	1	1	1	1					
A+1	M+1		1	1	0	1	1	1					
D-1			0	0	1	1	1	0					
A-1	M-1		1	1	0	0	1	0					
D+A	D+M		0	0	0	0	1	0					
D-A	D-M		0	1	0	0	1	1					
A-D	M-D		0	0	0	1	1	1					
D&A	D&M		0	0	0	0	0	0					
D A	D M		0	1	0	1	0	1					

jump				Эффект:			
j	j	j	j				
null	0	0	0	нет перехода			
JGT	0	0	1	if comp > 0 jump			
JEQ	0	1	0	if comp = 0 jump			
JGE	0	1	1	if comp ≥ 0 jump			
JLT	1	0	0	if comp < 0 jump			
JNE	1	0	1	if comp ≠ 0 jump			
JLE	1	1	0	if comp ≤ 0 jump			
JMP	1	1	1	безусловный переход			

a \ \ 0 a \ \ 1

Иллюстрация 4.5. Набор команд Nask с символическими мнемониками и соответствующими им двоичными кодами.

A-команда

A-команда записывает в регистр A некоторое 15-битное значение. Двоичная версия состоит из двух полей: кода операции (op-code), равного 0 (самый левый бит), за которым следуют пятнадцать битов, кодирующих неотрицательное двоичное число. Например, символьная команда @5 и ее двоичная версия 0000000000000101, сохраняет двоичное представление числа 5 в регистре A.

А-команда используется для трех различных целей. Во-первых, она обеспечивает единственный способ ввода константы в компьютер под управлением программы. Во-вторых, создает основу для последующей С-команды, которая будет выполнять действие с выбранным регистром ОЗУ, обозначаемым М, — записывает в регистр А адрес этого регистра М. В-третьих, создает основу для последующей С-команды, которая будет указывать переход, — записывает в регистр А адрес места перехода.

С-команда

С-команда отвечает на три вопроса: что вычислять (операция АЛУ, обозначается *compr*), где хранить вычисленное значение (*dest*) и что делать дальше (*jump*). Наряду с А-командой А, С-команда определяет все возможные операции компьютера.

В двоичном варианте крайний левый бит — это операционный код С-команды, который равен 1. Следующие два бита по соглашению не используются и равны 1. Следующие семь битов — двоичное представление поля *compr*. Следующие три бита — двоичное представление поля *dest*. Самые правые три бита — двоичное представление поля *jump*. Теперь опишем синтаксис и семантику этих трех полей.

Спецификация вычислений (*compr*): АЛУ компьютера Наск предназначено для вычисления одной из фиксированного набора функций для двух заданных 16-битных чисел. В компьютере Наск два 16-битных входа данных АЛУ подключены следующим образом. Первый вход АЛУ принимает данные из регистра D. Второй вход АЛУ принимает данные либо из регистра А (когда а-бит равен 0), либо из М, выбранного регистра памяти данных (когда а-бит равен 1). В совокупности вычисляемая функция задается а-битом и шестью с-битами, составляющими поле *compr* команды. Этот 7-битный шаблон потенциально может кодировать 128 различных функций, из которых только двадцать восемь, перечисленных на рисунке 4.5, документированы в спецификации языка.

Напомним формат С-команды: `111accccccddjjj`. Предположим, мы хотим вычислить значение регистра D минус 1. Согласно иллюстрации 4.5, это можно сделать с помощью символьной команды `D - 1`, которая в двоичном виде выглядит как `1110001110000000` (соответствующее 7-битное поле *comp* подчеркнуто для наглядности). Для вычисления побитовой операции `Or` над значениями регистров D и M используется команда `D | M` (в двоичном виде: `1111010101000000`). Для вычисления константы `-1` используется команда `-1` (в двоичном виде: `1110111010000000`) и т. д.

Спецификация назначения (*dest*): выходной сигнал АЛУ может сохраняться в одном из мест, а также одновременно в двух или трех возможных местах. Первый и второй d-биты определяют, сохранять ли вычисленное значение в регистре A и в регистре D соответственно. Третий d-бит определяет, сохранять ли вычисленное значение в M, то есть в выбранном в данный момент регистре памяти. Всего могут быть активизированы один, два, три или ни один из этих битов.

Напомним формат С-команды: `111accccccddjjj`. Предположим, мы хотим инкрементировать (увеличить на 1) значение регистра памяти с адресом 7, а также сохранить новое значение в регистре D. Согласно иллюстрации 4.5, это можно сделать с помощью двух команд:

```
0000000000000111 // @7
1111110111011000 // DM = M + 1
```

Указание перехода (*jump*): поле перехода в С-команде определяет дальнейшее действие. Есть две возможности: выбрать и выполнить следующую команду программы, то есть действие по умолчанию, или выбрать и выполнить какую-то другую указанную команду. Предполагается, что во втором случае регистр A уже содержит адрес нужной команды.

Во время выполнения команды необходимость перехода определяется совместно тремя j-битами поля перехода команды и выходом АЛУ. Первый, второй и третий j-биты определяют, следует ли переходить, если соответственно на выходе АЛУ отрицательное число, нуль

или положительное число. Всего получается восемь возможных условий перехода, перечисленных в правом нижнем углу иллюстрации 4.5. Инструкция безусловного перехода `goto` обозначается как 0; JMP (поскольку указывать поле *сotr* обязательно, то, согласно спецификации языка, оно вычисляется равным нулю в результате выполнения произвольно выбранной операции АЛУ, что игнорируется).

Предотвращение конфликтов использования регистра А: в компьютере Hack используется один и тот же адресный регистр для адресации как ОЗУ, так и ПЗУ. Таким образом, выполняя команду `@n`, компьютер одновременно выбирает и `RAM[n]`, и `ROM[n]`. Так задается основа для выполнения последующей С-команды, работающей с выбранным регистром памяти данных М, либо для выполнения последующей С-команды, задающей переход. Чтобы убедиться, что компьютер будет выполняет именно одну из этих двух операций, мы советуем придерживаться следующего правила: С-команда, содержащая ссылку на М, не должна содержать перехода, и наоборот: С-команда, задающая переход, не должна содержать ссылки на М.

4.2.4. Символы

Команды ассемблера могут задавать участки или ячейки (адреса) памяти с помощью констант или символов. Символы делятся на три функциональные категории: *предопределенные символы*, представляющие специальные адреса памяти; *символы меток*, представляющие места назначения команд перехода `goto`; и *символы переменных*, представляющие переменные.

Предопределенные символы: существует несколько видов предопределенных символов, предназначенных для повышения согласованности и читаемости низкоуровневых программ Hack.

R0, R1, ..., R15: эти символы привязаны к значениям от 0 до 15, что повышает читабельность программ Hack. Для примера рассмотрим следующий фрагмент кода:

```
// Set RAM[3] to 7 (Установить RAM[3] равным 7):
```

```
@7
```

```
D=A
```

```
@R3
```

```
M=D
```

Команда @7 записывает в регистр A значение 7, а команда @R3 записывает в регистр A значение 3. Почему мы в первом случае не используем R, а во втором используем? Потому что это делает код более понятным. Синтаксис команды @7 намекает на то, что A используется как регистр данных, а побочный эффект в виде выбора RAM[7] игнорируется. Синтаксис команды @R3 намекает на то, что A используется для выбора адреса памяти данных. В целом предопределенные символы R0, R1, ..., R15 можно рассматривать как готовые рабочие переменные, иногда называемые *виртуальными регистрами*.

SP, LCL, ARG, THIS, THAT: эти символы привязаны к значениям 0, 1, 2, 3 и 4 соответственно. Например, адрес 2 можно выбрать с помощью команд @2, @R2 или @ARG. Символы SP, LCL, ARG, THIS и THAT будут использоваться во второй части книги, когда мы реализуем компилятор и виртуальную машину, работающие поверх платформы Hack. Эти символы можно пока игнорировать; здесь мы указываем их для полноты спецификации.

SCREEN, KBD: программы Hack могут считывать данные с клавиатуры и выводить их на экран. Экран и клавиатура взаимодействуют с компьютером через два специально выделенных блока памяти, называемых *картами памяти*. Символы SCREEN и KBD связаны соответственно со значениями 16384 и 24576 (в шестнадцатеричном виде: 4000 и 6000) — это заранее условленные базовые адреса *карты памяти экрана* и *карты памяти клавиатуры*. Эти символы используются в программах Hack, которые управляют экраном и клавиатурой, как мы увидим в следующем разделе.

Символы меток: метки могут появляться в любом месте программы на языке ассемблера Nask, и объявляются они с помощью синтаксиса (*xxx*). Эта директива связывает символ *xxx* с адресом следующей команды в программе. Команды перехода *goto* с символами меток могут размещаться в любом месте программы, даже до объявления метки. По соглашению символы меток записываются заглавными буквами. В программе, приведенной на иллюстрации 4.4, используются три символа меток: *LOOP*, *STOP* и *END*.

Символы переменных: любой символ *xxx*, появляющийся в программе языка ассемблера Nask, не предопределенный и не объявленный в другом месте с помощью (*xxx*), рассматривается как переменная и привязывается к уникальному текущему числу (адресу памяти), начинающемуся с 16. По соглашению символы переменных записываются строчными буквами. Например, в программе, приведенной на иллюстрации 4.4, используются две переменные: *i* и *sum*. Ассемблер присваивает им значения 16 и 17 соответственно. Поэтому после трансляции команды *@i* и *@sum* выбирают участки памяти с адресами 16 и 17 соответственно. Преимущества такого решения заключаются в том, что в программе на машинном языке можно совершенно не обращать внимания на физические адреса. Можно использовать только символы, полагаясь на то, что ассемблер знает, как преобразовывать их в реальные адреса.

4.2.5. Обработка ввода/вывода

Аппаратную платформу Nask можно подключать к двум периферийным устройствам ввода/вывода: экрану и клавиатуре. Оба устройства взаимодействуют с компьютерной платформой через *карты памяти*.

Вывод пикселей на экран осуществляется посредством записи двоичных значений в определенный сегмент памяти, связанный с экраном, а ввод значений с клавиатуры — посредством считывания определенного участка памяти, связанного с клавиатурой. Физические устройства ввода/вывода и их карты памяти синхронизируются

посредством непрерывных циклов обновления, внешних по отношению к основной аппаратной платформе.

Экран: компьютер Nask взаимодействует с черно-белым экраном, организованным в виде 256 строк по 512 пикселей в строке. Содержимое экрана представлено картой памяти, хранящейся в блоке памяти объемом в 8К из 16-битных слов, начиная с адреса RAM 16384 (в шестнадцатеричном исчислении: 4000), также обозначаемого предопределенным символом SCREEN. Каждая строка физического экрана, начиная с левого верхнего угла экрана, представлена в оперативной памяти тридцатью двумя последовательными 16-битными словами. По соглашению началом экрана служит левый верхний угол, который считается строкой 0 и столбцом 0. Исходя из этого, пиксель в строке *row* и столбце *col* отображается битом номер $col \% 16$ (считая от правого, наименее значимого, к левому, наиболее значимому) слова, расположенного в $RAM[SCREEN + row \cdot 32 + col / 16]$. (Прим. переводчика: здесь знаки % и / означают остаток и неполное частное при делении.) Этот пиксель можно либо прочесть (узнать, черный он или белый), либо сделать черным, задав ему значение 1, либо сделать белым, задав ему значение 0. Рассмотрим в качестве примера фрагмент кода, затемняющий первые 16 пикселей в верхней левой части экрана:

```
// Записывает в регистр A адрес регистра RAM,  
// соответствующий 16 крайним левым пикселям верхней  
// строки экрана:
```

```
@SCREEN
```

```
// Записать в этот регистр RAM значение  
1111111111111111:
```

```
M= -1
```

Обратите внимание, что команды Nask не могут обращаться к отдельным пикселям/битам напрямую. Вместо этого мы должны

получить доступ к полному 16-битному слову из карты памяти, определить, каким битом или битами мы хотим манипулировать, выполнить манипуляцию с помощью арифметических/логических операций (не затрагивая другие биты), а затем записать измененное 16-битное слово в память. В приведенном выше примере мы обошлись без выполнения битовых манипуляций, поскольку задачу можно реализовать с помощью одной общей манипуляции.

Клавиатура: компьютер Hack может взаимодействовать со стандартной физической клавиатурой через однословную карту памяти, расположенную по адресу RAM 24576 (в шестнадцатеричном исчислении: 6000), также обозначаемую предопределенным символом KBD. Взаимодействие происходит следующим образом: когда на физической клавиатуре нажата клавиша, ее 16-битный код символа записывается в RAM[KBD]. Когда клавиша не нажата, в регистр записывается код 0. Список кодов символов Hack приведен в приложении 5.

Читатели с опытом программирования, вероятно, уже поняли, что манипулирование устройствами ввода/вывода с помощью языка ассемблера — дело весьма утомительное. Это потому, что они привыкли использовать операторы высокоуровневых языков программирования типа `write ("hello")` или `draw Circle (x, y, radius)`. Как вы теперь понимаете, существует значительный разрыв между этими абстрактными высокоуровневыми операциями ввода-вывода и побитовыми машинными командами, реализующими их для конкретных микросхем. Одним из средств устранения такого разрыва служит операционная система — программа, которая, помимо всего прочего, знает, как отображать текст и графику посредством манипуляций с пикселями. Во второй части книги мы обсудим и напишем одну такую ОС.

4.2.7. Синтаксические условности и форматы файлов

Файлы двоичного кода: по соглашению программы, написанные на двоичном языке Hack, хранятся в текстовых файлах с расширением `hack`, например, `Prog.hack`. Каждая строка в файле кодирует одну двоичную команду в виде последовательности шестнадцати символов

0 и 1. В совокупности все строки в файле представляют собой программу на машинном языке. Работает она следующим образом: когда программа на машинном языке загружается в память команд компьютера, двоичный код, записанный в n -й строке файла, сохраняется по адресу n в памяти команд. Счетчики строк программы, команд и адресов памяти принимают начальное значение 0.

Файлы на языке ассемблера: по соглашению программы, написанные на символьном языке ассемблера Hack, хранятся в текстовых файлах с расширением `asm`, например, `Prog.asm`. Файл на языке ассемблера состоит из текстовых строк, каждая из которых представляет собой А-команду, С-команду, объявление метки или комментарий.

Объявление метки — это текстовая строка вида (*символ*). Ассемблер обрабатывает такое объявление, связывая *символ* с адресом следующей команды в программе. Это единственное действие, которое выполняет ассемблер при обработке объявления метки; двоичный код при этом не генерируется. Именно поэтому объявления меток иногда называют *псевдокомандами (псевдоинструкциями)*: они существуют только на символьном уровне, не генерируя никакого кода.

Константы и символы: это символы `xxx` в А-командах в форме `@xxx`. Константы — это неотрицательные значения от 0 до $2^{15} - 1$, записываемые в десятичной системе счисления. Символ — это любая последовательность букв, цифр, символа подчеркивания (`_`), точки (`.`), знака доллара (`$`) и двоеточия (`:`), которая не начинается с цифры.

Комментарии: строка текста, начинающаяся с двух косых черт (`//`) и заканчивающаяся концом строки, считается комментарием и игнорируется.

Пробелы и пустые строки: пробелы перед командами и пустые строки игнорируются.

Регистр букв: все мнемоники ассемблера (иллюстрация 4.5) следует писать в верхнем регистре. По соглашению символы меток пишутся

в верхнем регистре, а символы переменных — в нижнем. Примеры см. на иллюстрации 4.4.

4.3. Программирование на языке Наск

Рассмотрим теперь три примера низкоуровневого программирования на языке ассемблера Наск. Поскольку проект 4 посвящен написанию программ на ассемблере Наск, вам будет полезно внимательно рассмотреть и понять эти примеры.

Пример 1: на иллюстрации 4.6 показана программа, которая складывает значения первых двух регистров ОЗУ, прибавляет к сумме 17 и сохраняет результат в третьем регистре ОЗУ. Перед запуском программы пользователь (или тестовый скрипт) должен поместить некоторые значения в RAM[0] и RAM[1].

```
// Program: Add.asm
// Computes: RAM[2] \ RAM[0] + RAM[1] + 17
// Usage: put values in RAM[0] and in RAM[1]
// D \ RAM[0]
@R0
D\M
// D \ D + RAM[1]
@R1
D\D+M
// D \ D + 17
@17
D\D+A
// RAM[2] \ D
@R2
M\D
(END)
@END
0; JMP
```

Иллюстрация 4.6. Программа на языке ассемблера Наск, вычисляющая простое арифметическое выражение.

Среди прочего программа иллюстрирует использование в качестве рабочих переменных так называемых виртуальных регистров $R0$, $R1$, $R2$... Она также иллюстрирует рекомендуемый способ завершения программ Наск, заключающийся во входе в установленный бесконечный цикл. Если бы такого цикла не было, то, согласно процессорной логике исполнения команд, компьютер продолжил бы усердно двигаться дальше и пытаться выполнять любые команды, которые могут храниться в памяти команд после последней команды текущей программы. Это может привести к непредсказуемым и потенциально опасным последствиям. Специально устанавливаемый бесконечный цикл служит средством контроля и сдерживания работы центрального процессора после завершения выполнения программы.

Пример 2: второй пример вычисляет сумму $1 + 2 + 3 + \dots + n$, где n — значение первого регистра ОЗУ, и помещает сумму во второй регистр ОЗУ. Эта программа была показана на иллюстрации 4.4, и теперь у нас есть все необходимое для ее полного понимания.

Помимо прочего, данная программа иллюстрирует использование символьных переменных — в данном случае i и sum . Этот пример также иллюстрирует рекомендуемую нами практику разработки низкоразрядных программ: вместо того чтобы писать непосредственно ассемблерный код, начните с написания псевдокода с *goto*-переходами. Затем проверьте свой псевдокод на бумаге, отслеживая значения ключевых переменных. Когда вы убедитесь в том, что логика программы верна и что она делает то, что должна делать, переходите к выражению каждой псевдокоманды в виде одной или нескольких команд на языке ассемблера.

На преимущества написания и отладки символьных (а не физических) команд обратила внимание еще в 1843 году писательница и одаренный математик Августа Ада Кинг-Ноэль, графиня Лавлейс. Это важное открытие поспособствовало становлению ее прочной славы как первого в истории программиста. До Ады Лавлейс составление программ для первых механических вычислительных устройств сводилось к непосредственному вмешательству в машинные операции,

так что кодирование было занятием трудным и подверженным многочисленным ошибкам. То, что было верно в 1843 году в отношении символьного и физического программирования, сегодня в той же степени верно в отношении псевдокода и языка ассемблера: когда речь идет о нетривиальных программах, писать и тестировать псевдокод, а затем переводить его в команды для ассемблера проще и безопаснее, чем писать код для ассемблера напрямую.

Пример 3: рассмотрим высокоуровневую идиому обработки массивов `for i = 0 ... n {делать что-то с arr[i]}`. Если мы хотим выразить данную логику на ассемблере, то наша первая проблема заключается в том, что в машинном языке не существует абстракции массива. Однако если нам известен базовый адрес массива в оперативной памяти, мы можем легко реализовать эту логику на языке ассемблера, используя доступ к элементам массива на основе указателей.

Для иллюстрации понятия указателя предположим, что переменная x содержит значение 523, и рассмотрим две возможные псевдокоманды $x = 17$ и $*x = 17$ (из которых мы выполним только одну). Первая команда присваивает переменной x значение 17. Вторая команда сообщает, что x следует рассматривать как указатель, то есть переменную, значение которой интерпретируется как адрес памяти. Следовательно, команда $*x = 17$ устанавливает в `RAM[523]` значение 17, оставляя само значение x нетронутым.

Программа на рисунке 4.7 иллюстрирует обработку массивов на основе указателей на машинном языке Hack. Главный интерес для нас представляют последовательные команды $A = D + M$ и $M = -1$. В языке Hack базовая идиома обработки указателей реализуется командой вида $A = \dots$, за которой следует C-команда, работающая с M (то есть с `RAM[A]`, участком памяти, выбранным по адресу A). Как мы увидим при написании компилятора во второй части книги, эта скромная идиома низкоуровневого программирования позволяет реализовать на языке ассемблера Hack любой доступ к массиву или объектно-ориентированную операцию *get/set*, выраженную на любом языке высокого уровня.

Псевдокод

```
// Программа: PointerDemo
// Начиная с базового адреса R0,
// присвоить первым R1 словам
// значение -1

n = 0
LOOP:
  if (n == R1) goto END
  *(R0 + n) = -1
  n = n + 1
  goto LOOP
END:
```

Язык ассемблера Hack

```
// Программа: PointerDemo.asm
// Начиная с базового адреса R0,
// присвоить первым R1 словам
// значение -1

    // n = 0
    @n
    M=0
(LLOOP)
    // if (n == R1) goto END
    @n
    D=M
    @R1
    D=D-M
    @END
    D;JEQ
    // *(R0 + n) = -1
    @R0
    D=M
    @n
    A=D+M
    M=-1
    // n = n + 1
    @n
    M=M+1
    // goto LOOP
    @LLOOP
    0;JMP
(END)
    @END
    0;JMP
```

Иллюстрация 4.7. Пример обработки массива с использованием доступа к элементам массива на основе указателя.

4.4. Проект

Задача: овладеть навыками низкоуровневого программирования и познакомиться с компьютерной системой Hack. Это будет сделано посредством написания и выполнения двух низкоуровневых программ на языке ассемблера Hack.

Ресурсы: единственные ресурсы, необходимые для выполнения проекта, — это эмулятор процессора Hack, расположенный в папке

nand2tetris/tools, и описанные ниже тестовые скрипты, расположенные в папке projects/04.

Контракт: напишите и протестируйте две описанные ниже программы. При выполнении на прилагаемом эмуляторе процессора ваши программы должны реализовать описанные модели поведения.

Умножение (Mult.asm): входными данными этой программы являются значения, хранящиеся в R0 и R1 (RAM[0] и RAM[1]). Программа вычисляет произведение $R0 * R1$ и сохраняет результат в R2. Предположим, что $R0 \geq 0$, $R1 \geq 0$ и $R0 * R1 < 32768$ (ваша программа не должна проверять эти высказывания). Поставляемые скрипты Mult.tst и Mult.cmp предназначены для проверки вашей программы на репрезентативных значениях данных.

Обработка ввода/вывода (Fill.asm): эта программа запускает бесконечный цикл считывания данных с клавиатуры. Когда нажата клавиша (любая), программа затемняет экран, делая каждый пиксель *черным*. Когда клавиша не нажата, программа очищает экран, делая каждый пиксель *белым*. Для затемнения и очищения экрана можно выбрать любую пространственную схему, если только достаточно долгое нажатие клавиши будет приводить к полному затемнению экрана, а достаточно долгое отсутствие нажатия — к полному очищению (осветлению) экрана. Для этой программы есть тестовый сценарий (Fill.tst), но нет файла сравнения — ее следует проверять, наблюдая за смоделированным экраном в эмуляторе ЦПУ.

Эмулятор ЦПУ: эта программа, доступная в папке nand2tetris/tools, осуществляет визуальное моделирование компьютера Nask (см. иллюстрацию 4.8). Графический интерфейс программы показывает текущее состояние памяти команд (ROM, ОЗУ), памяти данных (RAM, ПЗУ), двух регистров A и D, счетчика команд PC и АЛУ. Она также отображает текущее состояние экрана компьютера и позволяет вводить данные с клавиатуры.

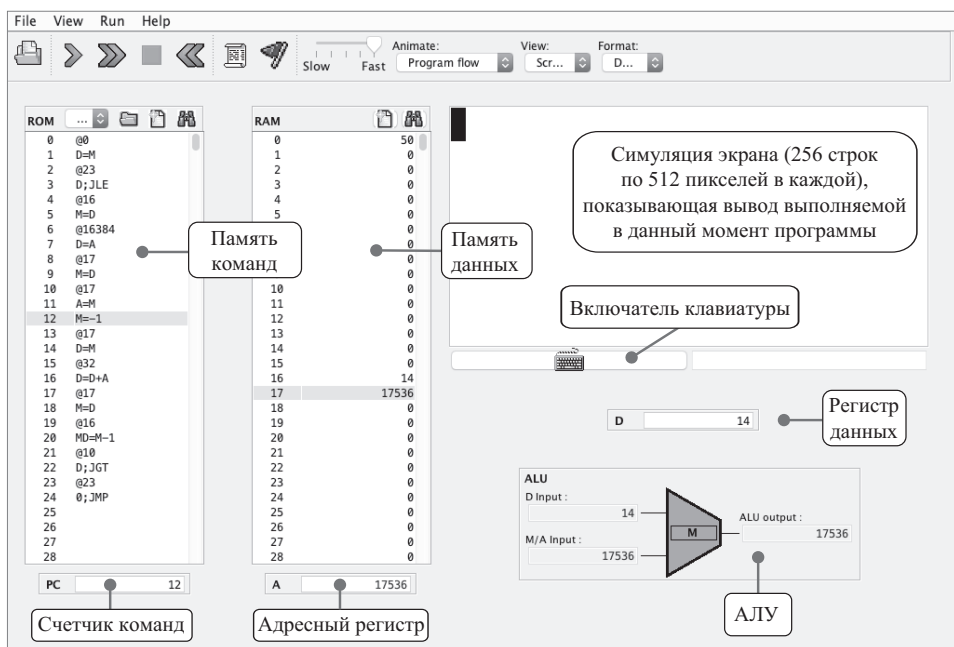


Иллюстрация 4.8. Эмулятор ЦПУ с загруженной в память команд (ROM) программой и с некоторыми данными, загруженными в память данных (RAM). На иллюстрации показан скриншот, сделанный во время выполнения программы.

Типичный способ использования эмулятора процессора — загрузить в ПЗУ программу на машинном языке, выполнить код и наблюдать за тем, как на него реагируют моделируемые аппаратные элементы. Важно отметить, что эмулятор процессора позволяет загружать двоичные файлы `.hack`, а также символьные файлы `.asm`, написанные на языке ассемблера Hack. В последнем случае эмулятор на лету переводит программу, записанную на языке ассемблера, в двоичный код. Удобно, что загруженный код можно просматривать как в двоичном, так и в символьном виде.

Поскольку поставляемый эмулятор процессора имеет встроенный ассемблер, в данном проекте нет необходимости использовать отдельный ассемблер Hack.

Шаги: мы рекомендуем следующий порядок действий.

0. Поставляемый эмулятор процессора располагается в папке `nand2tetris/tools`. Если вам нужна помощь, обратитесь к учебному руководству, доступному на сайте www.nand2tetris.org.
1. Напишите/отредактируйте программу `Mult.asm` с помощью обычного текстового редактора. Начните с каркасной программы, сохраненной в папке `projects/04/mult/Mult.asm`.
2. Загрузите `Mult.asm` в эмулятор процессора. Это можно сделать либо в интерактивном режиме, либо загрузив и выполнив прилагаемый скрипт `Mult.tst`.
3. Запустите скрипт. Если возникнут ошибки трансляции или времени выполнения, перейдите к шагу 1. Выполните шаги 1–3 для написания второй программы в папке `projects/04/fill`.

Совет по отладке: язык `Nack` чувствителен к регистру. Одна из распространенных ошибок программирования на языке ассемблера — это, например, запись `@foo` и `@Foo` в разных частях программы с ожиданием, что обе команды будут относиться к одному и тому же символу. В действительности же ассемблер сгенерирует две совершенно разные переменные, не имеющие между собой ничего общего.

Веб-версия проекта 4 доступна на сайте www.nand2tetris.org.

4.5. Перспектива

Язык `Nack` — это базовый пример машинного языка. Типичные машинные языки имеют больше операций, больше типов данных, больше регистров и больше форматов команд. Мы намеренно решили упростить синтаксис `Nack` и придать ему более легкий вид по сравнению с обычными языками ассемблера. В частности, мы выбрали дружественный синтаксис для `C`-команды, например `D = D + M`, вместо более распространенного префиксного синтаксиса `add M, D`, используемого во многих машинных языках. Читателю, однако, стоит

отметить, что это всего лишь особенность синтаксиса. Например, символ $+$ в коде операции $D + M$ не играет никакой алгебраической роли. Вся трехсимвольная строка $D + M$, взятая в целом, рассматривается как единая мнемоника ассемблера, предназначенная для кодирования одной операции АЛУ.

Одна из основных характеристик, придающих машинным языкам особый колорит, — количество адресов памяти, которые можно вместить в одной команде. В этом отношении строгий язык Наск можно описать как « $\frac{1}{2}$ -адресный». В одной 16-битной команде не хватает места для одновременной передачи кода команды и 15-битного адреса, и поэтому операции, связанные с доступом к памяти, требуют двух команд: одна для указания адреса, над которым мы хотим работать, и другая для указания операции. Для сравнения: многие машинные языки могут в каждой машинной команде указывать как операцию, так и по крайней мере один адрес.

И действительно, ассемблерный код Наск обычно представляет собой последовательность чередующихся А- и С-команд: @sum с последующим $M = 0$, @LOOP с последующим 0; JMP и т. д. Если вы считаете такой стиль кодирования утомительным или непривычным, то учтите, что в язык легко можно ввести более дружелюбные макрокоманды, такие как `sum = 0` и `goto LOOP`, что сделает ассемблерный код Наск короче и более читаемым. Для этого нужно будет всего лишь дополнить ассемблер средствами перевода каждой макрокоманды в две команды Наск, а это относительно простая задача.

Неоднократно упоминавшийся в данной главе *ассемблер* — это программа, отвечающая за преобразование символьных программ, записанных на языке ассемблера, в исполняемые программы, написанные в двоичном коде. Кроме того, ассемблер отвечает за управление всеми системными и пользовательскими символами, встречающимися в программе на языке ассемблера, и за их преобразование в физические адреса памяти, которые вводятся в сгенерированный двоичный код. Мы вернемся к этой задаче трансляции в главе 6, посвященной ассемблерам и их созданию.

5. Компьютерная архитектура

Делайте просто, насколько это возможно, но не проще.

— Альберт Эйнштейн (1879–1955)

Эта глава — вершина аппаратной части нашего путешествия. Теперь мы готовы взять микросхемы, которые собрали в главах 1–3, и интегрировать их в универсальную компьютерную систему, способную выполнять программы, написанные на машинном языке, представленном в главе 4. Конкретный компьютер, который мы построим и который называется Наск, имеет два важных достоинства. С одной стороны, Наск — это простая машина, которую можно собрать за несколько часов, используя ранее собранные микросхемы и прилагаемый аппаратный симулятор. С другой стороны, Наск достаточно мощен, чтобы проиллюстрировать основные принципы работы и аппаратные элементы любого компьютера общего назначения. Поэтому, собирая его, вы получите практические сведения о том, как работают современные компьютеры и как они устроены.

Раздел 5.1 начинается с обзора *архитектуры фон Неймана* — центральной догмы информатики, лежащей в основе конструкции почти всех современных компьютеров. Платформа Наск представляет собой вариант машины фон Неймана, и в разделе 5.2 приведена ее точная аппаратная спецификация. В разделе 5.3 описывается, как платформу Наск можно реализовать на основе ранее созданных микросхем, в частности АЛУ, созданного в проекте 2, а также регистров и устройств памяти, созданных в проекте 3. В разделе 5.4 описывается

проект, в рамках которого вы будете собирать компьютер. В разделе 5.5 представлены перспективы. В частности, мы сравниваем машину Nask с промышленными компьютерами и подчеркиваем критическую роль, которую в них играет оптимизация.

Компьютер, который появится в результате всех этих усилий, будет максимально простым, но не упрощенным. С одной стороны, он будет основан на минимальной и элегантной аппаратной конфигурации. С другой стороны, полученная конфигурация будет достаточно мощной для выполнения программ, написанных на Java-подобном языке программирования, представленном во второй части книги. Этот язык позволит разрабатывать интерактивные компьютерные игры и приложения с использованием графики и анимации, обеспечивая при этом высокую производительность с предоставлением удовлетворительного пользовательского опыта. Чтобы реализовать эти высокоуровневые приложения на «голой» аппаратной платформе, нам потребуется создать компилятор, виртуальную машину и операционную систему. Это мы сделаем в части II. Пока же завершим часть I, объединив микросхемы, которые мы создавали до сих пор, в полноценную аппаратную платформу общего назначения.

5.1. Основы компьютерной архитектуры

5.1.1. Принцип хранимой программы

По сравнению со всеми другими окружающими нас машинами, самая замечательная особенность цифрового компьютера — это его удивительная универсальность. Компьютер — это машина с ограниченным и фиксированным аппаратным обеспечением, которая может выполнять бесконечное количество задач, от игр до набора текста книг и управления автомобилем. Его поразительная универсальность (преимущество, которое мы считаем само собой разумеющимся) — в действительности плод блестящей идеи под названием «принцип хранимой программы», независимо сформулированной несколькими учеными и инженерами на заре развития вычислительной техники в 1930-х годах. Этот

принцип до сих пор считается самым важным изобретением в современной компьютерной науке, если не самой ее основой.

Как и во многих других научных открытиях, основная идея проста. Компьютер основан на фиксированной аппаратной платформе, способной выполнять ограниченный набор простых команд. В то же время эти команды можно комбинировать, как строительные блоки, создавая произвольно сложные программы. Более того, логика данных программ не встроена в аппаратное обеспечение, как это было принято для механических вычислительных устройств до 1930-х годов. Вместо этого код программы временно хранится в памяти компьютера, *как и данные*, и становится тем, что сейчас называется «*программным обеспечением*». Поскольку работа компьютера проявляется для пользователя через исполняемое в данный момент программное обеспечение, одну и ту же аппаратную платформу можно заставить вести себя совершенно по-разному, загружая в нее каждый раз другую программу.

5.1.2. Архитектура фон Неймана

Принцип хранимой программы служит ключевым элементом как абстрактных, так и практических компьютерных моделей, в первую очередь *машины Тьюринга* (1936 г.) и *машины фон Неймана* (1945 г.). Машина Тьюринга — абстрактная модель, описывающая обманчиво простой компьютер, и эта модель используется в основном в теоретической информатике для анализа логических основ вычислений. В отличие от нее, машина фон Неймана представляет собой практическую модель, на основе которой строятся почти все современные компьютерные платформы.

Архитектура фон Неймана, показанная на иллюстрации 5.1, основана на *центральной процессорной устройстве (ЦПУ)*, которое взаимодействует с устройством *памяти*, получающим данные от некоторого устройства *ввода*, и выдает данные на некоторое устройство *вывода*. В основе этой архитектуры лежит концепция *хранимой программы*: в памяти компьютера хранятся не только данные, которыми компьютер манипулирует, но и команды, которые указывают компьютеру, что делать. Рассмотрим эту архитектуру подробнее.

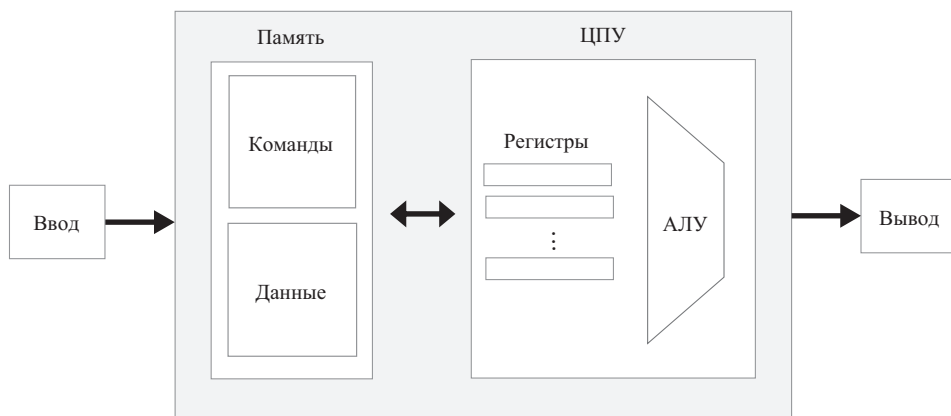


Иллюстрация 5.1. Общая компьютерная архитектура фон Неймана.

5.1.3. Память

Память компьютера можно рассматривать как с физической, так и с логической точки зрения. Физически память представляет собой линейную последовательность адресуемых *регистров* фиксированного размера, каждый из которых имеет уникальный адрес и значению. Логически это адресное пространство служит двум целям: хранение данных и хранению команд. Как «слова команд», так и «слова данных» реализованы одинаково — в виде последовательности битов.

Все регистры памяти, независимо от их роли, обрабатываются одинаково: чтобы получить доступ к определенному регистру памяти, мы указываем его адрес. Это действие, также называемое *адресацией*, обеспечивает немедленный доступ к данным регистра. Термин «*память с произвольным доступом*» (*random access memory, RAM*) происходит от того важного требования, что доступ к каждому случайно выбранному регистру памяти должен осуществляться мгновенно, то есть в течение одного цикла (или временного шага), независимо от объема памяти и расположения регистра. Это требование явно бесполезно для устройств памяти, имеющих миллиарды регистров. Читатели, которые собирали устройства RAM (ОЗУ) в проекте 3, знают, что мы уже выполнили данное требование.

В дальнейшем мы будем называть область памяти, предназначенную для хранения данных, *памятью данных*, а область памяти, предназначенную для хранения команд (машинных инструкций), *памятью команд*. В некоторых вариантах архитектуры фон Неймана области для памяти данных и памяти команд выделяются и обрабатываются динамически, по мере необходимости, в одном и том же физическом адресном пространстве. В других вариантах память данных и память команд хранятся в двух физически отдельных блоках памяти, каждый из которых имеет свое собственное адресное пространство. Оба варианта имеют свои плюсы и минусы, о чем мы поговорим позже.

Память данных: программы высокого уровня предназначены для манипулирования абстрактными артефактами, такими как переменные, массивы и объекты. Однако на аппаратном уровне эти абстракции данных реализуются в виде двоичных значений, хранящихся в регистрах памяти. В частности, после перевода на машинный язык абстрактная обработка массивов и операции *get/set* над объектами сводятся к *чтению* и *записи* выбранных регистров памяти. Чтобы прочитать хранимые в регистре данные, мы указываем адрес и узнаем значение выбранного регистра. Для записи в регистр мы указываем адрес и сохраняем новое значение в выбранном регистре, перезаписывая его предыдущее значение.

Память команд: прежде чем программа высокого уровня будет выполнена на целевом компьютере, ее нужно перевести на машинный язык этого компьютера. Каждое высказывание языка высокого уровня преобразуется в одну или несколько команд низкого уровня, которые затем записываются в виде двоичных значений в файл, называемый *двоичной*, или *исполняемой*, версией программы. Прежде чем запустить программу, мы должны сначала загрузить ее двоичную версию с устройства хранения данных и направить в виде последовательности команд в *память команд* компьютера.

С точки зрения архитектуры компьютера то, *как* программа загружается в память компьютера, считается внешним вопросом. Важно то,

что, когда центральному процессору поручают выполнить программу, код программы должен уже находиться в памяти компьютера.

5.1.4. Центральное процессорное устройство

Центральное процессорное устройство (ЦПУ), или просто центральный процессор — это центральный элемент архитектуры компьютера, ответственный за выполнение команд текущей программы. Каждая команда указывает ЦПУ, какие вычисления нужно выполнить, к каким регистрам обратиться, какую команду извлечь и выполнить следующей. Центральный процессор выполняет эти задачи с помощью трех основных элементов: арифметико-логического устройства (АЛУ), набора регистров и блока управления.

Арифметико-логическое устройство: микросхема ALU предназначена для выполнения всех доступных компьютеру низкоуровневых арифметических и логических операций. Типичное АЛУ может складывать два заданных значения, осуществлять над ними побитовую операцию And (И), проверять их на равенство и т. д. Насколько функциональным должно быть АЛУ — это решение, принимаемое для каждого проекта отдельно. В целом любую функцию, не поддерживаемую АЛУ, можно реализовать позже с помощью системного программного обеспечения, работающего поверх аппаратной платформы. Компромисс прост: аппаратные реализации обычно более эффективны, но приводят к удорожанию оборудования, в то время как программные реализации недороги и менее эффективны.

Регистры: в процессе выполнения вычислений процессору часто требуется временно хранить промежуточные значения. Теоретически эти значения можно было бы хранить в регистрах памяти, но такое решение повлекло бы за собой слишком частую передачу данных между центральным процессором и оперативной памятью, которые представляют собой две отдельные микросхемы, расположенные на некотором расстоянии друг от друга. Задержки, связанные с передачей

данных, замедляли бы работу находящегося в ЦПУ АЛУ, представляющего собой сверхбыстрый комбинационный вычислитель. В результате возникала бы ситуация так называемого *информационного голода*, или зависания, которое происходит, когда быстрый процессор зависит от медленного хранилища данных, не успевающего передавать данные на его входы.

Чтобы избежать информационного голода и повысить производительность компьютера, процессор обычно оснащается небольшим набором высокоскоростных (и относительно дорогих) *регистров*, выполняющих роль оперативной памяти непосредственно самого процессора. Эти регистры служат для различных целей: *регистры данных* хранят промежуточные значения, *адресные регистры* хранят значения, которые используются для обращения к адресам оперативной памяти, *счетчик команд* хранит адрес инструкции, которая должна быть извлечена и выполнена следующей, а *регистр команд* хранит текущую команду. Типичный процессор использует несколько десятков таких регистров, но нашему экономному компьютеру Hack понадобятся всего три.

Контроль: компьютерная команда — это структурированный набор заранее условленных микрокодов, то есть последовательностей из одного или нескольких битов, предназначенных для передачи сигналов различным устройствам о том, что нужно делать. Таким образом, прежде чем компьютер выполнит команду, ее нужно сначала декодировать в микрокоды. Затем каждый микрокод направляется к определенному аппаратному устройству (АЛУ, регистры, память) в ЦПУ, где он сообщает устройству, как участвовать в общем выполнении команды.

Выборка — исполнение: на каждом шаге (цикле) выполнения программы ЦПУ извлекает двоичную машинную команду из памяти команд, декодирует ее и выполняет. В качестве побочного эффекта выполнения команды ЦПУ также выясняет, какую команду следует извлечь и выполнить следующей. Этот повторяющийся цикл иногда называют *циклом выборки — исполнения*.

5.1.5. Ввод и вывод

Компьютеры взаимодействуют с внешней средой с помощью множества различных устройств ввода и вывода (*input/output, I/O*) — это экраны, клавиатуры, устройства хранения данных, принтеры, микрофоны, динамики, сетевые карты и т. д. (не говоря уже о бесчисленных датчиках и активаторах, встроенных в автомобили), а также камеры, слуховые аппараты, системы сигнализации и разнообразные гаджеты вокруг нас. Мы здесь не обращаем внимания на все эти устройства ввода-вывода по двум причинам. Во-первых, каждое из них представляет собой уникальный механизм, требующий уникальных инженерных знаний. Во-вторых, по этой же причине специалисты по компьютерным технологиям разработали хитроумные схемы абстрагирования от этой сложности и придания всем устройствам ввода-вывода одинакового вида для компьютера. Ключевым элементом данной абстракции служит так называемый *ввод-вывод с привязкой к памяти* (или *с отображением на память*).

Основная идея заключается в том, чтобы создать эмуляцию устройства ввода-вывода в двоичном виде, в результате чего оно будет восприниматься процессором как обычный линейный сегмент памяти. При этом для каждого устройства ввода-вывода в определенной области памяти компьютера выделяется свой сегмент — так называемая *карта памяти*. В случае устройства ввода, такого как клавиатура, карта памяти постоянно отражает физическое состояние устройства: когда пользователь нажимает клавишу на клавиатуре, в карте памяти клавиатуры появляется двоичный код, соответствующий этой клавише. В случае устройства вывода, например экрана, это устройство постоянно отражает состояние своей карты памяти: когда в карту памяти экрана записывается бит, на экране включается или выключается соответствующий пиксель.

Устройства ввода/вывода и карты памяти обновляются, или синхронизируются, много раз в секунду, поэтому время отклика с точки зрения пользователя кажется мгновенным. С точки зрения программирования ключевой момент здесь в том, что низкоуровневые

компьютерные программы могут получить доступ к любому устройству ввода/вывода, манипулируя назначенной ему картой памяти.

Соглашение о привязке устройств к памяти основано на нескольких принципах. Во-первых, данные, управляющие каждым устройством ввода-вывода, должны быть сериализованы, или последовательно отображены в памяти компьютера — отсюда и название «карта памяти» (иногда ее называют «картой распределения памяти»). Например, экран, представляющий собой двумерную сетку пикселей, отображается на одномерный блок регистров памяти фиксированного размера. Во-вторых, каждое устройство ввода-вывода должно поддерживать установленный протокол взаимодействия, чтобы программы могли обращаться к нему предсказуемым образом. Например, двоичные коды, присвоенные тем или иным клавишам на клавиатуре, должны быть согласованы и иметь только одно значение. При том что компьютерных платформ существует великое множество, а устройств ввода-вывода различных производителей аппаратного и программного обеспечения и того больше, остается только поражаться тому, какую огромную роль в реализации низкоуровневого взаимодействия устройств сыграли согласованные отраслевые *стандарты*.

Практические последствия ввода-вывода с привязкой к памяти весьма значительны: компьютерная система совершенно не зависит от количества, характера или моделей устройств ввода-вывода, которые взаимодействуют или могут взаимодействовать с ней. Каждый раз, когда мы хотим подключить к компьютеру новое устройство ввода-вывода, все, что нам нужно сделать, — это выделить ему новую карту распределения памяти и записать базовый адрес карты (эти одноразовые конфигурации выполняются так называемыми *программами-установщиками*). Другой необходимый элемент — программа-драйвер устройства, добавляемая в операционную систему компьютера. Эта программа служит связующим звеном между данными карты памяти устройства ввода-вывода и тем, как эти данные фактически отображаются на физическом устройстве ввода-вывода или генерируются им.

5.2. Аппаратная платформа Наск: спецификация

Описанная выше архитектурная структура характерна для любой универсальной компьютерной системы, или компьютера общего назначения. Теперь мы перейдем к описанию одного конкретного варианта этой архитектуры: компьютера Наск. Как это принято в путешествии «От Nand до “Тетриса”», начнем мы с абстракции, сосредоточившись на том, *что* должен делать компьютер. Реализация компьютера — *как* он это делает — будет описана позже.

5.2.1. Обзор

Платформа Наск — это 16-разрядная машина фон Неймана, предназначенная для выполнения программ, написанных на машинном языке Наск. В связи с этим платформа Наск состоит из *центрального процессора (ЦПУ)*, двух отдельных модулей памяти, выполняющих функции *памяти команд* и *памяти данных*, и двух устройств ввода-вывода с привязкой к памяти: *экрана* и *клавиатуры*.

Компьютер Наск выполняет программы, находящиеся в памяти команд. В физических реализациях платформы Наск память инструкций может быть реализована в виде микросхемы *ПЗУ* (постоянного запоминающего устройства, или памяти, доступной только для чтения, *read-only memory*, ROM), в которую предварительно загружена необходимая программа. Программные эмуляторы компьютера Наск поддерживают эту функциональность благодаря средствам загрузки памяти команд из текстового файла, содержащего программу, написанную на машинном языке Наск.

Центральный процессор Наск состоит из АЛУ, построенного в проекте 2, и трех регистров — *регистра данных (D)*, *адресного регистра (A)* и *счетчика команд* (программного счетчика PC). Регистр D и регистр A идентичны микросхеме Register, построенной в проекте 3, а счетчик команд идентичен микросхеме PC, построенной в проекте 3. В то время как регистр D используется исключительно для хранения значений данных, регистр A служит для одной из трех различных

целей — в зависимости от контекста, в котором он используется: хранения значения данных (подобно регистру D), выбора адреса в памяти команд или выбора адреса в памяти данных.

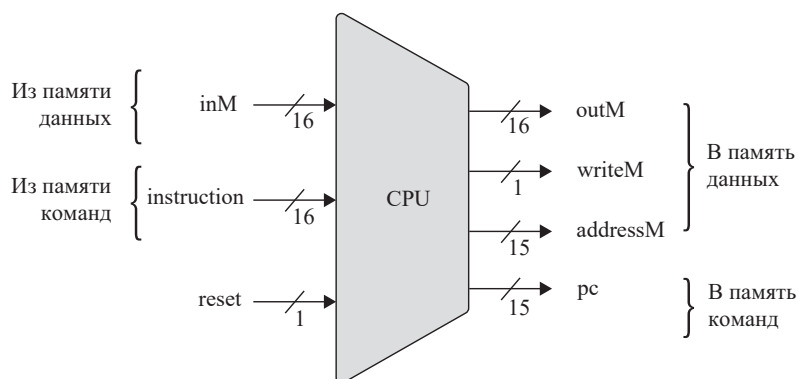
Центральный процессор Hack предназначен для выполнения команд, написанных на машинном языке Hack. В случае A-команды 16 бит команды рассматриваются как двоичное значение, загружаемое в регистр A в том виде, в каком оно есть. В случае C-команды инструкция рассматривается как набор управляющих битов, которые определяют различные микрооперации, выполняемые различными частями микросхемы внутри процессора. Перейдем теперь к описанию того, как центральный процессор материализует эти микрокоды в конкретные действия.

5.2.2. Центральное процессорное устройство

Интерфейс центрального процессора Hack показан на иллюстрации 5.2. Центральный процессор предназначен для выполнения 16-битных команд в соответствии со спецификацией машинного языка Hack, представленной в главе 4.

Центральный процессор состоит из АЛУ, двух регистров A и D и программного счетчика PC (эти внутренние части микросхемы не видны снаружи). Центральный процессор ожидает подключения к памяти команд, из которой он получает команды для выполнения, и к памяти данных, из которой он может читать и в которую может записывать значения данных. Вход `inM` и выход `outM` содержат значения, обозначаемые в синтаксисе C-команд буквой M. Выход `addressM` содержит адрес, по которому должно быть записано значение `outM`.

Если на вход `instruction` подается A-команда, то ЦПУ загружает 16-битное значение команды в регистр A. Если на вход `instruction` подается C-команда, то ЦПУ: 1) заставляет АЛУ выполнить указанное командой вычисление и 2) сохраняет это значение в подмножестве {A, D, M} регистров назначения, указанных в команде. Если среди регистров назначения указан M, на выход `outM` процессора подается значение с выхода АЛУ, а на выход `writeM` подается сигнал 1. В противном случае на выход `writeM` подается сигнал 0, а на выходе `outM` может появиться любое значение.



Chip name: CPU

Input:

```
instruction[16] // Команда на выполнение
inM[16]         // ввoд M для команды (содержимое RAM[A])
reset           // сигнал перезагрузки программы (если reset==1)
                // или продолжения выполнения программы (если reset==0).
```

Output:

```
outM[16]        // Запись в RAM[addressM], вывод инструкции M
addressM[15]    // По какому адресу записывать?
writeM          // Записать в память?
pc[15]          // Адрес следующей команды
```

Иллюстрация 5.2. Интерфейс центрального процессорного устройства (ЦПУ) компьютера Hack.

Пока на входе `reset` подается сигнал 0, ЦПУ на основании вывода АЛУ и битов перехода текущей команды определяет, какую команду выбрать следующей. Если на `reset` подается сигнал 1, процессор подает на выход `pc` сигнал 0. Позже в этой главе мы подключим выход `pc` процессора к адресному входу микросхемы памяти команд, чтобы та выдавала следующую команду. Такая конфигурация реализует этап выборки в цикле «выборка — исполнение».

Выходы `outM` и `writeM` процессора реализованы с помощью комбинационной логики; так выполняемые команды влияют на них мгновенно. Выходы `addressM` и `pc` синхронизируемые: хотя на них и влияют выполняемые команды, новые значения фиксируются на них только на следующем временном шаге.

5.2.3. Память команд

Спецификация памяти команд Nask под названием ROM32 показана на иллюстрации 5.3.



Chip name: ROM32K

Input: address[15]

Output: out[16]

Function: Выводит 16-битное значение, хранимое по адресу, указанному на входе address. Предполагается, что в микросхему заранее загружена программа, написанная на машинном языке Nask.

Иллюстрация 5.3. Интерфейс памяти команд Nask.

5.2.4. Ввод/вывод

Доступ к устройствам ввода/вывода компьютера Nask осуществляется посредством *памяти данных* компьютера, представляющей собой устройство ОЗУ с возможностью чтения/записи, состоящее из 32К адресуемых 16-битных регистров. Помимо того что память данных служит хранилищем данных общего назначения, она также служит интерфейсом между центральным процессором и устройствами ввода/вывода компьютера, о чем мы сейчас и поговорим.

Платформу Nask можно подключать к двум периферийным устройствам: *экрану* и *клавиатуре*. Оба эти устройства взаимодействуют с компьютерной платформой через определенные области памяти, называемые *картами памяти*. В частности, изображения рисуются на экране посредством записи 16-битных значений в определенный сегмент памяти, называемый *картой памяти экрана*. Аналогично клавишу, нажимаемую в данный момент на клавиатуре, можно

определить, обратившись к 16-битному регистру памяти, называемому *картой памяти клавиатуры*.

Карты памяти экрана и клавиатуры много раз в секунду обновляются внешней по отношению к компьютеру периферийной логикой обновления. Таким образом, когда в карте памяти экрана изменяются один или несколько битов, это изменение немедленно отражается на физическом экране. Аналогично, когда на физической клавиатуре нажимается клавиша, код символа нажатой клавиши немедленно появляется в карте памяти клавиатуры. В соответствии с этим, когда низкоуровневая программа должна прочесть что-то с клавиатуры или записать что-то на экран, она манипулирует соответствующими картами памяти этих устройств ввода-вывода.

В компьютерной платформе Nask карта памяти экрана и карта памяти клавиатуры реализуются двумя встроенными микросхемами, названными Screen и Keyboard. Эти микросхемы ведут себя как стандартные устройства памяти, но с дополнительным побочным эффектом непрерывной синхронизации между устройствами ввода-вывода и соответствующими картами памяти. Рассмотрим их подробнее.

Экран: компьютер Nask может взаимодействовать с физическим экраном, состоящим из 256 рядов по 512 черно-белых пикселей в каждом, охватывающих сетку из 131 072 пикселей. Компьютер взаимодействует с физическим экраном через карту памяти, реализованную 8-килобайтной микросхемой памяти с 16-битными регистрами. Эта микросхема, названная Screen, ведет себя как обычная микросхема памяти, то есть из нее можно читать значения и записывать на нее значения с помощью обычного интерфейса RAM. Кроме того, чип Screen обладает дополнительным эффектом: состояние любого из его битов постоянно отражается соответствующим пикселем на физическом экране (1 = черный, 0 = белый).

Физический экран представляет собой двумерное адресное пространство, где каждый пиксель соотносится с определенными строкой и столбцом. Языки программирования высокого уровня обычно содержат графическую библиотеку, позволяющую обращаться

к отдельным пикселям посредством задания координат (*строка, столбец*). Однако карта памяти, представляющая этот двумерный экран на низком уровне, в действительности является одномерной последовательностью 16-битных слов, каждое из которых идентифицируется посредством указания адреса. Как следствие, получить прямой доступ к отдельным пикселям невозможно. Вместо этого нужно выяснить, в каком слове находится требуемый бит, а затем получить доступ ко всему 16-битному слову, в которое входит данный бит, и манипулировать всем словом. Точное соотношение между этими двумя адресными пространствами и конкретная привязка к памяти указаны на иллюстрации 5.4. Привязка экрана к памяти будет реализована драйвером экрана операционной системы, которую мы разработаем во второй части книги.

```

Chip name: Screen      // Карта памяти экрана
Input:      in[16]      // Что записывать
            address[13] // Откуда читать/куда записывать
            load        // Бит разрешения
Output:     out[16]     // Значение Screen по указанному адресу.
Function:    Такая же, как у 16-битного 8K RAM, плюс дополнительный эффект
            обновления экрана.

```

Выдает значение, хранящееся в ячейке памяти, указанной адресом. Если `load==1`, в указанной адресом ячейке устанавливается значение `in`.

Загруженное значение подается на выход `out` со следующего временного шага и далее. Кроме того, чип постоянно обновляет физический экран, состоящий из 256 строк и 512 столбцов черно-белых пикселей.

Пиксель в строке `r` сверху и столбце `c` слева ($0 \leq r \leq 255$, $0 \leq c \leq 511$) привязан к биту $c \% 16$ (считая от наименее значимого бита до наиболее значимого бита) 16-битного слова, хранящегося в `Screen[r * 32 + c / 16]`.

(Ожидается, что симуляторы компьютера Hack будут имитировать физический экран, привязку к памяти и постоянное обновление.)

Иллюстрация 5.4. Интерфейс микросхемы Screen компьютера Hack.

Клавиатура: компьютер Hack может взаимодействовать с физической клавиатурой, такой же как у персонального компьютера. Компьютер взаимодействует с физической клавиатурой через карту памяти, реализуемую микросхемой Keyboard, интерфейс которой показан на иллюстрации 5.5. Интерфейс микросхемы идентичен интерфейсу

16-битного регистра, доступного только для чтения. Кроме того, микросхема `Keyboard` имеет дополнительный эффект отражения состояния физической клавиатуры: когда на физической клавиатуре нажата клавиша, на выходе `out` микросхемы `Keyboard` появляется 16-битный код соответствующего символа. Когда клавиша не нажата, микросхема выдает 0. Набор символов, поддерживаемых компьютером `Nack`, приведен в приложении 5 вместе с кодом каждого символа.

Chip name: `Keyboard` // Карта памяти клавиатуры

Output: `out[16]`

Function: Выдает 16-битный код символа клавиши, нажатой в текущий момент на физической клавиатуре, либо 0, если никакая клавиша не нажата.

(Ожидается, что симуляторы компьютера `Nack` будут имитировать постоянное обновление.)

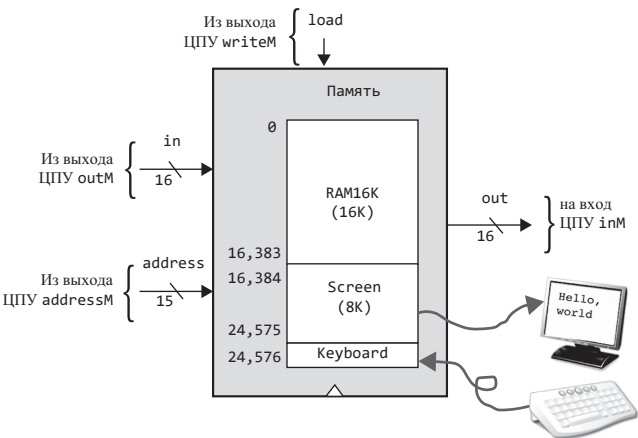
Иллюстрация 5.5. Интерфейс микросхемы `Keyboard` компьютера `Nack`.

5.2.5. Память данных

Общее адресное пространство *памяти данных* `Nack` реализуется микросхемой под названием `Memory`. Эта микросхема представляет собой пакет из трех 16-битных частей: `RAM16K` (микросхема ОЗУ на 16K регистров, служащая хранилищем данных общего назначения), `Screen` (встроенная микросхема ОЗУ на 8K регистров, служащая картой памяти экрана) и `Keyboard` (встроенная микросхема регистра, служащая картой памяти клавиатуры). Полная спецификация приведена на иллюстрации 5.6.

5.2.6. Компьютер

Самый верхний элемент в иерархии аппаратного обеспечения `Nack` — это чип (условная микросхема) под названием `Computer` (иллюстрация 5.7). `Computer` можно подключить к экрану и клавиатуре. Пользователь видит экран, клавиатуру и однобитное значение входа `reset`. Когда пользователь подает на него сигнал 1, а затем 0, компьютер начинает выполнять текущую загруженную программу. С этого момента пользователь взаимодействует исключительно с программным обеспечением.



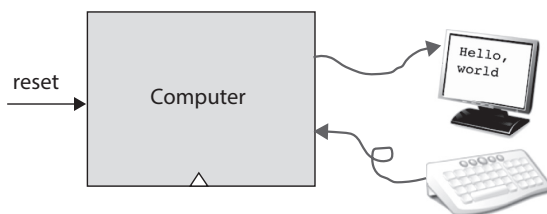
```
Chip name: Memory           // Память данных
Input:      in[16]           // Что писать
            address[15]      // Откуда читать/куда писать
            load             // Бит разрешения записи
Output:     out[16]          // Значение по указанному адресу
Function:
```

Полное адресное пространство памяти данных компьютера Hack.
Используются только верхние 16K+8K+1 слов адресного пространства.
Адреса в диапазоне 0-16383 предоставляют доступ к RAM16K.
Адреса в диапазоне 16384-24575 предоставляют доступ к Screen.
Адрес 24576 предоставляет доступ к Keyboard.
Другие адреса недоступны.

Иллюстрация 5.6. Интерфейс памяти данных Hack. Обратите внимание на то, что десятичным значениям 16384 и 24576 соответствуют шестнадцатеричные 4000 и 6000.

Такая логика запуска реализует то, что иногда называют *загрузкой* компьютера. Когда вы, например, загружаете ПК или мобильный телефон, устройство настраивается на выполнение программы, находящейся в ПЗУ. Эта программа, в свою очередь, загружает ядро операционной системы (также являющееся программой) в оперативную память и запускает его. Затем ядро запускает процесс (еще одну программу), считывающий данные с устройств ввода компьютера, то есть клавиатуры, мыши, сенсорного экрана, микрофона и т. д. В какой-то момент пользователь совершает какое-то действие, и ОС отвечает на него запуском другого процесса или вызовом какой-либо программы.

В компьютере Наск программное обеспечение состоит из двоичной последовательности 16-битных инструкций, записанных на машинном языке Наск и хранящихся в памяти команд компьютера. Обычно этот двоичный код представляет собой низкоуровневую версию программы, написанной на каком-либо языке высокого уровня и переведенной компилятором на машинный язык Наск. Процесс компиляции будет описан и реализован во второй части книги.



Chip name: Computer

Input: reset

Function:

Когда `reset==0`, выполняется хранящаяся в компьютере программа.

Когда `reset==1`, программа перезапускается.

Для запуска программы установить значение `reset` равным 1, затем 0.

(Предполагается, что в память команд компьютера уже загружена программа, написанная на машинном языке Наск.)

Иллюстрация 5.7. Интерфейс самой верхней в иерархии микросхемы Computer аппаратной платформы Наск.

5.3. Реализация

В этом разделе описывается аппаратная реализация компьютера Наск, спецификация которого описана выше. Как обычно, мы не даем точных инструкций по сборке. Скорее ожидаем, что читатели сами найдут способы и придумают некоторые детали реализации. Все описанные ниже микросхемы можно построить на языке HDL и смоделировать на персональном компьютере с помощью прилагаемого симулятора аппаратуры.

5.3.1. Центральное процессорное устройство

Реализация процессора Hack подразумевает создание архитектуры логических вентилях, способной: 1) выполнять заданную команду Hack и 2) определять, какая команда должна быть извлечена и выполнена следующей. Для декодирования текущей команды мы будем использовать логические вентили, для вычисления указанной в команде функции — арифметико-логическое устройство (АЛУ), для хранения значения результата — набор регистров, а для отслеживания порядка инструкций — счетчик команд. Поскольку все основные строительные блоки (АЛУ, регистры, ПК и элементарные логические вентили) уже были построены в предыдущих главах, остается только придумать, как соединить эти части между собой так, чтобы реализовать предполагаемую функциональность ЦПУ. Одна из возможных конфигураций показана на иллюстрации 5.8 и объясняется ниже.

Декодирование команды: начнем со входа ЦПУ *instruction*. Подаваемое на него 16-битное значение представляет собой либо А-команду (когда крайний левый бит равен 0), либо С-команду (когда крайний левый бит равен 1). В случае А-команды биты инструкции интерпретируются как двоичное значение, которое следует загрузить в регистр А. В случае С-команды эта команда рассматривается как набор управляющих битов `1xxaccccccdddjjj` следующим образом: биты *a* и *cccccc* кодируют часть *comp* инструкции; биты *ddd* кодируют часть *dest* инструкции; биты *jjj* кодируют часть *jump* инструкции. Биты *xx* игнорируются.

Выполнение команды: в случае А-команды 16 бит команды загружаются в регистр А как есть (фактически это 15-битное значение, так как самый левый бит — это код операции 0). В случае С-команды *a*-бит определяет, будет ли на вход АЛУ подаваться значение из регистра А или значение *M* со входа *inM*. Биты *cccccc* определяют, какую именно функцию выполняет АЛУ. Биты *ddd* определяют, куда следует передавать вычисленное АЛУ значение. Биты *jjj* используются для определения того, какую команду выбирать следующей.

Архитектура процессора должна извлечь описанные выше управляющие биты из сигнала команды на входе `instruction` и направить их на входы тех микросхем-частей, где они участвуют в выполнении команды. Учтите, что каждая из этих микросхем-частей уже спроектирована для выполнения своей функции. Поэтому проектирование процессора сводится в основном к такому соединению существующих микросхем, которое реализует эту модель выполнения.

Выбор команды: в качестве побочного эффекта выполнения текущей команды ЦПУ определяет и выдает адрес команды, которая должна быть извлечена и выполнена следующей. Ключевой элемент данной подзадачи — *счетчик команд* (*Program Counter, PC*) — часть процессора, роль которой заключается в том, чтобы всегда хранить адрес следующей команды.

Согласно спецификации компьютера Hack, текущая программа хранится в памяти команд, начиная с адреса 0. Следовательно, если мы хотим запустить (или перезапустить) программу, нам нужно установить счетчик программ на 0. Вот почему на иллюстрации 5.8 вход `reset` центрального процессора соединен непосредственно со входом `reset` микросхемы PC. Если мы подадим сигнал 1 на этот бит, то получим значение $PC = 0$, что заставит компьютер получить и выполнить первую команду в программе. Если значение на входе `reset` было вновь установлено на 0, то счетчик выполняет операцию по умолчанию: $PC++$.

Но что, если текущая программа содержит директиву перехода? Согласно спецификации языка, в таком случае осуществляется переход к команде, адрес которой является текущим значением A. Таким образом, в реализации процессора нужно предусмотреть следующее поведение счетчика программы: если *jump*, то $PC = A$, иначе $PC++$.

Как добиться такого поведения с помощью логических вентилей? Намек на это показан на иллюстрации 5.8.

Обратите внимание, что на один из входов регистра PC поступает значение с выхода регистра A. Таким образом, если активировать вход `load` регистра PC, будет выполнена операция $PC = A$, а не операция

по умолчанию PC++. Это происходит только в том случае, если нужно выполнить переход. Возникает следующий вопрос: как именно мы узнаем о том, что нужно совершить переход? Это зависит от трех j-битов текущей команды и двух выходных битов АЛУ zr и ng. Вместе эти биты определяют, выполнено ли условие перехода или нет.

На этом мы остановимся, чтобы не лишать читателей удовольствия самим завершить реализацию процессора. Надеемся, что при этом они по достоинству оценят элегантность процессора Nask.

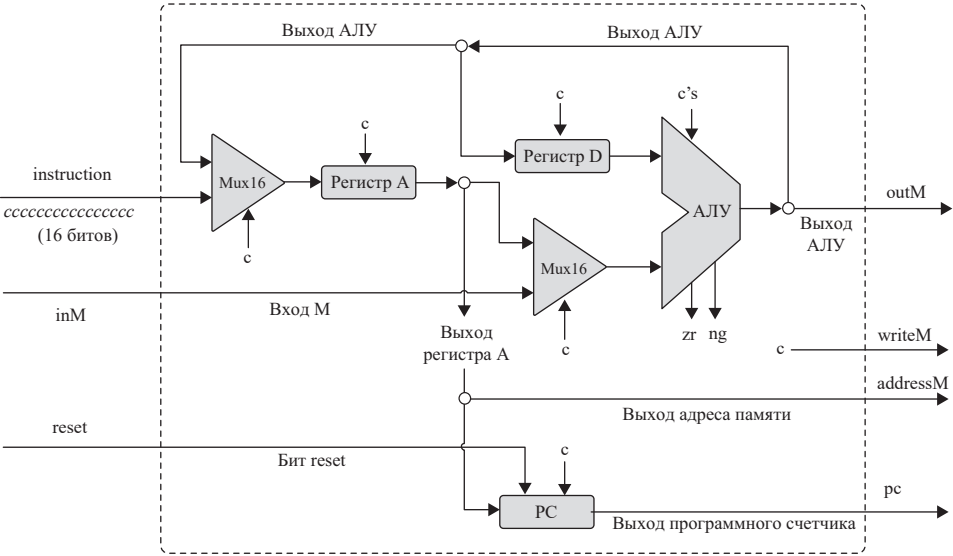


Иллюстрация 5.8. Предполагаемая реализация ЦПУ Nask, показывающая входящую 16-битную инструкцию (команду). Здесь инструкция обозначена символами cccccccccccccccc, чтобы подчеркнуть, что в случае C-команды она рассматривается как набор управляющих битов, предназначенных для управления различными частями микросхемы процессора. На этой диаграмме каждый символ c на входе какой-либо составной части процессора означает некоторый управляющий бит, извлеченный из инструкции (в случае АЛУ c' s означает шесть управляющих битов, указывающих АЛУ, что вычислять). В целом эти управляющие биты заставляют процессор выполнять указанную команду. Мы специально не показываем, куда идут конкретные биты, потому что хотим, чтобы читатели сами ответили на эти вопросы.

5.3.2. Память

Микросхема памяти компьютера Nask представляет собой совокупность трех микросхем: RAM16K, Screen и Keyboard. Однако эта модульность неявная: программы на машинном языке Nask видят *единое адресное пространство*, начиная с адреса 0 и заканчивая адресом 24576 (в шестнадцатеричном исчислении 6000).

Интерфейс микросхемы Memory показан на рисунке 5.6. В реализации этого интерфейса должен быть предусмотрен описанный выше эффект континуума. Например, если на вход address микросхемы Memory поступает значение 16384, реализация должна получить доступ к адресу 0 в микросхеме Screen, и т. д. И снова мы предпочитаем не приводить слишком много деталей и предоставить вам разобраться с остальной реализацией самим.

5.3.3. Компьютер

Итак, мы подошли к концу нашего путешествия по конструкции аппаратного обеспечения. Самый верхний в иерархии чип Computer можно реализовать с помощью трех микросхем: центрального процессора CPU, микросхемы памяти данных Memory и микросхемы памяти команд ROM32K. Подробности показаны на иллюстрации 5.9.

Реализация Computer разработана с учетом следующего цикла выборки — исполнения: когда пользователь активирует вход reset, на выход ЦПУ pc выводится значение 0, что заставляет память команд (ROM32K) подать на выход первую команду в программе. ЦПУ выполняет команду, причем это выполнение может подразумевать чтение или запись в регистр памяти данных. В процессе выполнения команды центральный процессор выясняет, какую команду следует выбрать следующей, и передает этот адрес через свой выход pc. С выхода pc ЦПУ значение поступает на вход address памяти команд, заставляя память подавать на выход команду, которая должна быть выполнена следующей. Оттуда команда поступает на вход instruction центрального процессора, завершая цикл выборки — исполнения.

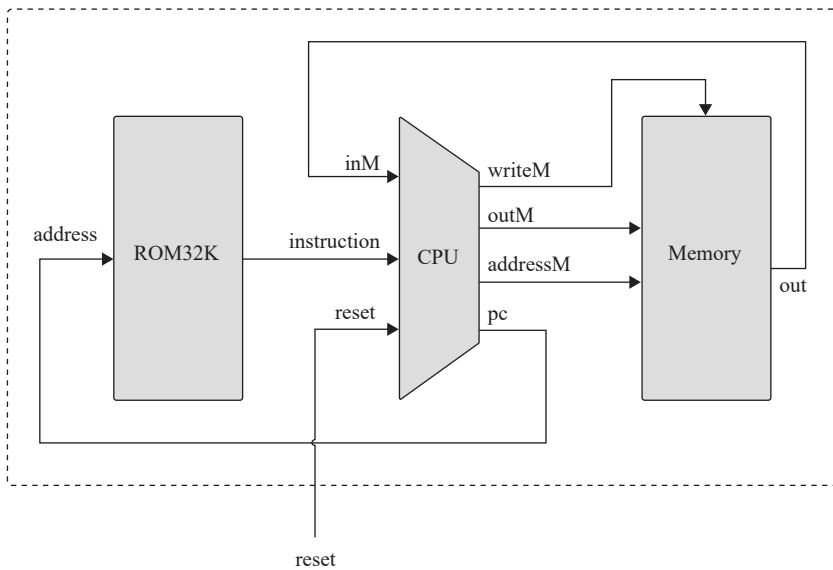


Иллюстрация 5.9. Предлагаемая реализация самого главного в иерархии платформы Hack чипа Computer.

5.4. Проект

Задача: завершить сборку компьютера Hack, самой верхней в иерархии которого является микросхема Computer.

Ресурсы: все микросхемы, описанные в этой главе, должны быть описаны на языке HDL и протестированы на прилагаемом симуляторе аппаратуры с использованием упомянутых ниже тестовых программ.

Контракт: создать аппаратную платформу, способную выполнять программы, написанные на машинном языке Hack. Продемонстрировать работу платформы, запустив для чипа Computer три прилагаемые тестовые программы.

Тестовые программы: естественный способ тестирования в целом реализации микросхемы Computer — это выполнение ею примеров

программ, написанных на машинном языке Hack. Для проведения такого теста можно написать тестовый сценарий, который загружает в аппаратный симулятор микросхему Computer, загружает программу из внешнего текстового файла в микросхему-часть ROM32K (память команд), а затем запускает генератор тактовой частоты на количество циклов, достаточное для выполнения программы. Мы предоставляем три такие тестовые программы, а также соответствующие тестовые сценарии и файлы сравнения.

- `Add.hack`: складывает две константы 2 и 3 и записывает результат в `RAM[0]`.
- `Max.hack`: вычисляет максимальное значение из `RAM[0]` и `RAM[1]` и записывает результат в `RAM[2]`.
- `Rect.hack`: рисует на экране прямоугольник из `RAM[0]` строк по 16 пикселей каждая. Верхний левый угол прямоугольника расположен в левом верхнем углу экрана.

Перед тестированием микросхемы вашего компьютера с помощью любой из этих программ просмотрите относящийся к этой программе тестовый сценарий и убедитесь, что вы понимаете отдаваемые симулятору команды. При необходимости обратитесь к приложению 3 («Язык описания тестов»).

Шаги: реализуйте компьютер в следующем порядке.

Memory: эту микросхему можно построить по приведенной на иллюстрации 5.6 общей схеме с использованием трех микросхем-частей: `RAM16K`, `Screen` и `Keyboard`. `Screen` и `Keyboard` доступны как встроенные чипы; собирать их нет необходимости. Хотя микросхема `RAM16K` была собрана в проекте 3, мы рекомендуем использовать ее встроенную версию.

CPU: центральный процессор можно построить по предложенной реализации на иллюстрации 5.8. В принципе, по мере необходимости можно воспользоваться построенным в проекте 2 АЛУ, построенными

в проекте 3 Register и PC и построенными в проекте 1 логическими вентилями. Но мы рекомендуем использовать встроенные версии всех этих микросхем (в частности, использовать встроенные регистры ARegister, DRegister и PC). Встроенные микросхемы имеют точно такую же функциональность, как и микросхемы, созданные в предыдущих проектах, но они заодно реализуют дополнительные эффекты графического интерфейса, облегчающие моделирование и тестирование.

В процессе реализации процессора у вас может возникнуть соблазн спроектировать и собрать собственные внутренние («вспомогательные») микросхемы. Имейте в виду, что в этом нет необходимости — процессор Hack можно вполне элегантно и эффективно реализовать с помощью только тех микросхем, что показаны на иллюстрации 5.8, с добавлением некоторых элементарных логических вентилях, построенных в проекте 1 (лучше всего использовать их встроенные версии).

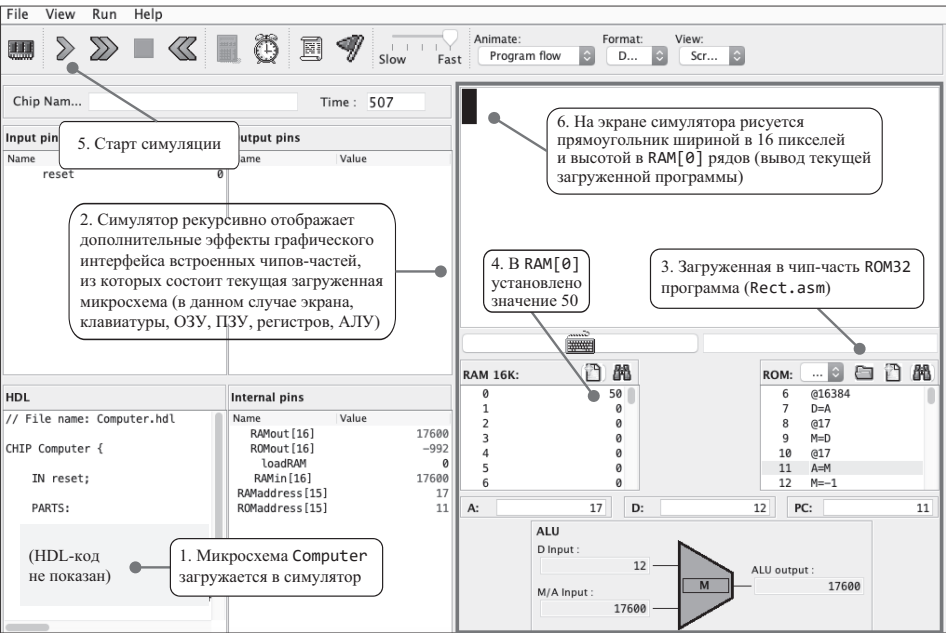


Иллюстрация 5.10. Тестирование микросхемы Computer на предлагаемом симуляторе аппаратуры. Хранимая в памяти программа Rect рисует в левом верхнем углу экрана прямоугольник из RAM[0] рядов по 16 пикселей черного цвета каждый.

Память команд: воспользуйтесь встроенной версией чипа ROM32.

Computer: компьютер можно построить согласно схеме, показанной на иллюстрации 5.9.

Симулятор аппаратуры: все микросхемы данного проекта, включая самую верхнюю микросхему Computer, можно реализовать с помощью прилагаемого симулятора аппаратуры. На иллюстрации 5.10 показан скриншот тестирования программы Rect.hack на реализации микросхемы Computer.

Веб-версия проекта 5 доступна по адресу: www.nand2tetris.org.

5.5. Перспектива

В соответствии с духом практического курса «От Nand до “Тетриса”» архитектура компьютера Hack минимальна. Реальные компьютерные платформы имеют больше регистров, больше типов данных, более мощные АЛУ и более богатые наборы команд. Однако большинство этих различий — количественные. С качественной точки зрения почти все цифровые компьютеры, включая Hack, основаны на одной и той же концептуальной архитектуре: машине фон Неймана.

По своему назначению компьютерные системы можно разделить на две категории: *компьютеры общего назначения* (универсальные) и *компьютеры специального назначения* (специализированные). Компьютеры общего назначения, такие как персональные компьютеры и сотовые телефоны, обычно взаимодействуют с пользователем. Они предназначены для выполнения множества программ и легкого перехода от одной программы к другой. Специализированные компьютеры обычно встраиваются в другие системы, такие как автомобили, камеры, медиацентры, медицинские устройства, промышленные контроллеры и т. д. Для любого конкретного приложения в ПЗУ (память, предназначенную только для чтения) специализированного компьютера записывается одна программа. Например, в некоторых игровых

приставках игровое программное обеспечение находится во внешнем картридже, который представляет собой сменный модуль ПЗУ, заключенный в специфическую упаковку. Хотя компьютеры общего назначения обычно более сложны, чем специализированные, и позволяют выполнять большее количество программ, все они основаны на общих базовых архитектурных принципах: принципе хранимой программы, логики выборки — исполнения, идее центрального процессора, регистров и счетчиков.

Большинство компьютеров общего назначения используют одно адресное пространство для хранения программ и данных. Другие компьютеры, например Наск, используют два отдельных адресных пространства. Последняя конфигурация, которая по историческим причинам называется *гарвардской архитектурой*, менее гибкая с точки зрения возможностей ситуативного использования памяти, но имеет явные преимущества. Во-первых, она проще и дешевле в изготовлении. Во-вторых, часто быстрее, чем конфигурация с одним адресным пространством. Наконец, если размер программы, которую должен выполнить компьютер, известен заранее, размер памяти команд можно соответствующим образом оптимизировать и зафиксировать. По этим причинам гарвардская архитектура является предпочтительной во многих специализированных, узконаправленных и встраиваемых компьютерах.

Компьютеры, использующие одно и то же адресное пространство для хранения команд и данных, сталкиваются со следующей проблемой: как подать адрес команды и адрес регистра данных, с которым должна работать команда, на один и тот же адресный вход устройства общей памяти? Очевидно, одновременно сделать это нельзя. Стандартное решение основано на двухцикловой логике. Во время *цикла выборки* на адресный вход памяти подается адрес команды, что заставляет память немедленно выдать текущую команду, которая затем сохраняется в регистре инструкций. В последующем *цикле исполнения* команда декодируется, а на тот же адресный вход памяти подается адрес данных, с которыми должна работать эта команда. Другие же компьютеры, использующие отдельную память команд и данных (такие как Наск), выигрывают от одноцикловой логики выборки — исполнения,

которая быстрее и проще в работе. Платой за это служит необходимость использования отдельных блоков памяти данных и команд, хотя при этом нет необходимости использовать отдельный регистр инструкций.

Компьютер Наск взаимодействует с экраном и клавиатурой. Компьютеры общего назначения обычно подключены к многочисленным устройствам ввода-вывода, таким как принтеры, устройства хранения данных, сетевые карты и т. д. Кроме того, реальные дисплеи намного сложнее экрана компьютера Наск, они имеют больше пикселей, больше цветов и более высокую скорость отображения. Тем не менее основной принцип, согласно которому каждый пиксель управляется хранящимся в памяти двоичным значением, остается неизменным, только вместо одного бита, управляющего черным или белым цветом пикселя, на управление уровнем яркости каждого из нескольких основных цветов обычно отводится по 8 бит, и вместе они определяют конечный цвет пикселя. В результате получаются миллионы возможных цветов — больше, чем может различить человеческий глаз.

Реализация привязки экрана Наск к основной памяти компьютера также является упрощенной. Центральные процессоры многих компьютеров не управляют напрямую пикселями через биты памяти, а посылают высокоуровневые команды вроде «нарисовать линию» или «нарисовать круг» на отдельные графические микросхемы или отдельный графический процессор (GPU). Аппаратное и низкоуровневое программное обеспечение этих специализированных графических процессоров оптимизировано для обработки графики, анимации и видео, что снимает с центрального процессора и главного компьютера бремя непосредственной обработки этих прожорливых задач.

Наконец, следует подчеркнуть, что большая часть творческих и технических усилий в сфере проектирования компьютерного оборудования направлена на достижение лучшей производительности. Многие архитекторы аппаратного обеспечения посвящают свою работу ускорению доступа к памяти с использованием умных алгоритмов кэширования и структур данных, оптимизации доступа к устройствам ввода-вывода, реализации конвейерного принципа обработки и параллельных вычислений с использованием предварительной выборки

команд и других методов оптимизации, о которых не было сказано ни слова в этой главе.

Исторически попытки повысить производительность обработки данных привели к появлению двух основных лагерей в сфере разработки процессоров. Сторонники архитектуры с полным набором команд (*Complex Instruction Set Computing, CISC*) утверждали, что более высокая производительность достигается за счет создания более мощных процессоров с более сложными наборами команд. И наоборот, сторонники архитектуры с сокращенным набором команд (*Reduced Instruction Set Computing, RISC*) создавали более простые процессоры с более ограниченными наборами команд, утверждая, что такие процессоры действительно обеспечивают более высокую производительность в эталонных тестах. Компьютер Hack не имеет к этим дебатам никакого отношения, поскольку не имеет ни обширного набора команд, ни специальных аппаратных методов ускорения.

6. Ассемблер

Что в имени? То, что зовем мы розой, и под другим названием
сохраняло б свой сладкий запах!

— Шекспир, «Ромео и Джульетта»

В предыдущих главах мы завершили разработку аппаратной платформы, предназначенной для выполнения программ на машинном языке Nask. Мы представили две версии данного языка — символьную и двоичную — и сказали, что символьные программы можно переводить в двоичный код с помощью программы, называемой *ассемблером*. В этой главе мы опишем, как работают ассемблеры и как они строятся, что подведет нас к созданию ассемблера Nask — программы, которая переводит программы, написанные на символьном языке Nask, в двоичный код, который может выполняться на аппаратном обеспечении Nask.

Поскольку связь между символьными инструкциями и соответствующими им двоичными кодами проста, реализация ассемблера с помощью языка программирования высокого уровня не представляет собой сложной задачи. Одна из сложностей возникает из-за того, что в программах на языке ассемблера могут использоваться символические ссылки на адреса памяти. Ассемблер должен обрабатывать эти символы и преобразовывать их в физические адреса памяти. Данная задача обычно решается с помощью *таблицы символов* — широко используемой структуры данных.

Реализация ассемблера — первый из серии семи проектов по разработке программного обеспечения, сопровождающих вторую часть книги. Разработка ассемблера вооружит вас базовым набором общих

навыков, которые пригодятся вам во всех этих проектах и не только: работа с аргументами командной строки, работа с входными и выходными текстовыми файлами, синтаксический анализ команд, работа с пробелами, работа с символами, генерация кода и многие другие приемы используются во многих проектах по разработке программного обеспечения. Если у вас нет опыта программирования, вы можете разработать ассемблер на бумаге. Этот вариант описан в веб-версии проекта 6, доступной по адресу: www.nand2tetris.org.

6.1. Общие принципы

Машинные языки обычно задаются в двух вариантах: двоичном и символьном. Двоичная команда, например 110000100000001100000000000000111, представляет собой пакет микрокодов, установленный для декодирования и исполнения некоторой целевой аппаратной платформой. Например, самые левые 8 битов команды, 11000010, могут соответствовать операции типа «загрузка». Следующие 8 битов, 00000011, могут, допустим, представлять регистр R3. Оставшиеся 16 битов, 00000000000000111, могут, например, представлять значение 7. Когда мы приступаем к созданию аппаратной архитектуры и машинного языка, то можем установить правила, согласно которым эта конкретная 32-битная инструкция заставит аппаратное обеспечение выполнить операцию «загрузить константу 7 в регистр R3». Современные компьютерные платформы поддерживают сотни таких возможных операций. Как следствие, машинные языки бывают сложными, включающими в себя множество кодов операций, режимов адресации памяти и форматов команд.

Понятно, что запись программ в двоичном коде — настоящее мучение. Естественным решением напрашивается использование некоего символьного синтаксиса, соответствующего двоичным командам, например, «load R3, 7». Коды операции вроде load («загрузка») иногда называют мнемониками, что в переводе с латыни означает «метод запоминания». Поскольку перевод от мнемоник и символов к двоичному коду прост, имеет смысл писать низкоуровневые программы в символьной нотации, после чего компьютерная программа

переводит их в двоичный код. Символьный язык называется *языком ассемблера*, а собственно программа-переводчик, или «транслятор» — *ассемблером*. Ассемблер разбивает каждую символьную команду на ее базовые поля, например `load, R3 и 7`, переводит каждое поле в эквивалентный двоичный код и, наконец, собирает сгенерированные биты в двоичную команду, которая может выполняться аппаратным обеспечением. Отсюда и название «ассемблер» («сборщик»).

Символы: рассмотрим символьную команду `goto 312`. После трансляции эта команда заставляет компьютер получить и выполнить команду, хранящуюся по адресу 312, которая может быть началом некоторого цикла. Но если это начало цикла, то почему бы не отметить данную точку в языке ассемблера какой-нибудь описательной меткой, скажем, `LOOP` («петля, цикл»), и не использовать команду `goto LOOP` вместо `goto 312`? Для этого всего лишь нужно где-то записать, что `LOOP` означает 312. Правда, во время перевода в двоичный код каждое слово `LOOP` нужно будет заменить на 312, но это небольшая цена за выигрыш в читабельности и переносимости программы.

В целом символы в языках ассемблера используются для трех целей.

- *Установка меток:* в программах на языках ассемблера можно объявлять и использовать символы, которые отмечают различные места в коде, например, `LOOP` и `END`.
- *Объявление переменных:* в программах на языках ассемблера можно объявлять и использовать символические переменные, например, `i` и `sum`.
- *Передача заранее установленных значений:* в программах на языках ассемблера можно ссылаться на специальные адреса в памяти компьютера с помощью заранее установленных символов, например, `SCREEN` и `KBD`.

Конечно, за удобство приходится платить. Кто-то же должен отвечать за управление всеми этими символами. В частности, кто-то должен помнить, что `SCREEN` означает 16384, `LOOP` означает 312, `sum`

означает какой-то другой адрес и т. д. Эта задача по обработке символов — одна из самых важных функций ассемблера.

Пример: на иллюстрации 6.1 приведены две версии одной и той же программы, написанной на машинном языке Наск. Символьная версия включает в себя разнообразные элементы, которые так нравятся людям в компьютерных программах: комментарии, пробелы, отступы, символьные инструкции и символьные ссылки. Но ни один из этих элементов не имеет никакого смысла для компьютеров, которые понимают только одно: биты. Посредник, помогающий преодолеть разрыв между удобным для человека символическим кодом и понятным компьютеру — это ассемблер.

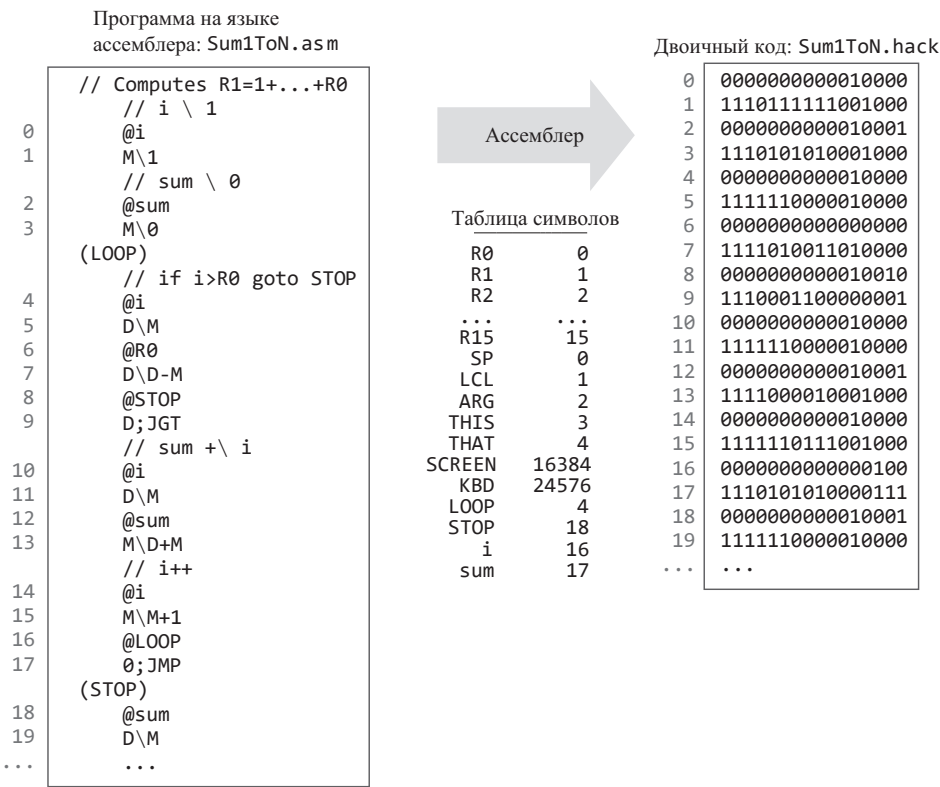


Иллюстрация 6.1. Код программы переводится в двоичный код с использованием таблицы символов. Номера строк, не являющиеся частью кода, указаны для более удобного просмотра.

Проигнорируем пока все подробности на иллюстрации 6.1, как и таблицу символов, и сделаем несколько общих замечаний. Во-первых, отметим, что, хотя номера строк и не являются частью кода, они играют важную, хотя и неявную роль в процессе трансляции. Если двоичный код будет загружаться в память команд, начиная с адреса 0, то номер строки каждой команды будет совпадать с ее адресом в памяти. Очевидно, что это наблюдение должно быть важным для разработчика ассемблера. Во-вторых, обратите внимание, что комментарии и объявления меток не генерируют никакого кода, поэтому их иногда называют *псевдоинструкциями*. И наконец, констатируя очевидное, отметим, что для того, чтобы написать ассемблер для некоторого машинного языка, разработчик ассемблера должен получить полную спецификацию символьного и двоичного синтаксиса языка.

Запомним это и перейдем к спецификации машинного языка Nask.

6.2. Спецификация машинного языка Nask

Язык ассемблера Nask и его эквивалентное двоичное представление были описаны в главе 4. Спецификация языка повторяется здесь для удобства. Эта спецификация — тот «контракт», который составители ассемблера Nask должны реализовать тем или иным способом.

6.2.1. Программы

Двоичная программа Nask: двоичная программа Nask — это последовательность текстовых строк, каждая из которых состоит из шестнадцати символов 0 и 1. Если строка начинается с 0, то она представляет собой двоичную A-команду. В противном случае она представляет собой двоичную C-команду.

Программа на языке ассемблера Nask: программа на языке ассемблера Nask — это последовательность текстовых строк, состоящих из команд ассемблера, объявлений меток или комментариев.

Предопределенные символы: в любой ассемблерной программе Hack разрешается использовать следующие предопределенные символы. R0, R1, ..., R15 означают 0, 1... 15 соответственно. SP, LCL, ARG, THIS, THAT означают 0, 1, 2, 3, 4 соответственно. SCREEN и KBD означают 16384 и 24576 соответственно. Значения этих символов интерпретируются как адреса в Hack RAM.

Символы меток: псевдокоманда (*xxx*) определяет символ *xxx* для ссылки на участок ПЗУ Hack, содержащий следующую команду программы. Символ метки можно определить один раз и использовать в любом месте программы на языке ассемблера, даже перед строкой, в которой он определен.

Символы переменных: любой символ *xxx*, появляющийся в программе на языке ассемблера, не предопределенный и не объявленный в другом месте меткой (*xxx*), рассматривается как переменная. Переменные сопоставляются с последовательными участками ОЗУ по мере их появления, начиная с адреса 16. Таким образом, первая переменная, встречающаяся в программе, запишется в RAM[16], вторая — в RAM[17] и т. д.

6.2.3. Соглашения о синтаксисе

Символы: символом может быть любая не начинающаяся с цифры последовательность букв, цифр, знака подчеркивания (), точки (.), знака доллара (\$) и двоеточия (:).

Константы: могут появляться только в A-командах в виде @*xxx*. Константа *xxx* представляет собой значение в диапазоне 0–32767 и записывается в десятичной системе счисления.

Пробелы и пустые строки: первые пробелы перед символами и пустые строки игнорируются.

Соглашение о регистре: все мнемоники ассемблера (такие как A + 1, JEQ и т. д.) должны записываться в верхнем регистре. Остальные

символы — метки и имена переменных — чувствительны к регистру. Рекомендуется использовать верхний регистр для меток и нижний регистр для переменных.

На этом спецификация машинного языка Nask завершена.

6.3. Перевод с языка ассемблера в двоичный код

В этом разделе описывается перевод программ, написанных на языке ассемблера Nask, в двоичный код. Сосредоточимся на разработке ассемблера для языка Nask, хотя описываемые нами методы применимы к любому ассемблеру.

Ассемблер принимает на вход поток команд на языке ассемблера и генерирует на выходе поток переведенных (транслированных) двоичных инструкций. Полученный код можно загрузить в память компьютера, и он будет исполнен. Чтобы выполнить процесс перевода (трансляции), ассемблер должен как-то обрабатывать команды и символы.

6.3.1. Обработка команд

Для каждой команды, написанной на языке ассемблера, ассемблер:

- разбирает команду на поля, из которых она состоит;
- для каждого поля генерирует соответствующий битовый код, как указано на иллюстрации 6.2;
- если команда содержит символическую ссылку, преобразует символ в его числовое значение;
- собирает полученные бинарные коды в строку из шестнадцати символов 0 и 1;
- записывает собранную строку в выходной файл.

6.3.2. Обработка символов

В ассемблерных программах разрешается использовать символьные метки (места назначения инструкций `goto`) до того, как символы будут

определены. Такая условность облегчает жизнь составителям ассемблерного кода и усложняет жизнь разработчикам ассемблера. Распространенное решение — разработка *двухпроходного ассемблера*, который читает код дважды, от начала до конца. На первом проходе ассемблер создает *таблицу символов*, добавляет в нее все символы меток, но не генерирует код. Во время второго прохода ассемблер обрабатывает символы переменных и генерирует двоичный код, используя таблицу символов. Далее приведены некоторые подробности.

Инициализация: ассемблер создает таблицу символов и инициализирует ее со всеми предопределенными символами и их предварительными распределенными значениями. На иллюстрации 6.1 результатом этапа инициализации служит таблица символов со всеми символами до KBD включительно.

Первый проход: ассемблер проходит через всю ассемблерную программу, строка за строкой, отслеживая номер строки. Этот номер начинается с 0 и увеличивается на 1 каждый раз, когда встречается А-команда или С-команда, но не изменяется, когда встречается комментарий или объявление метки. Каждый раз, когда встречается объявление метки (*xxx*), ассемблер добавляет новую запись в таблицу символов, связывая символ *xxx* с номером текущей строки плюс 1 (это будет ROM-адрес следующей инструкции в программе). В результате такого прохода в таблицу символов добавляются все символы меток программы вместе с их соответствующими значениями. На рисунке 6.1 первый проход приводит к добавлению в таблицу символов LOOP и STOP. Во время первого прохода код не генерируется.

Второй проход: ассемблер снова проходит через всю программу и разбирает каждую строку следующим образом. Каждый раз, когда встречается А-команда с символической ссылкой, а именно @*xxx*, где *xxx* — символ, а не число, ассемблер ищет *xxx* в таблице символов. Если символ найден, ассемблер заменяет его числовым значением и завершает трансляцию инструкции. Если символ не найден, то он должен представлять новую переменную.

Для ее обработки ассемблер: 1) добавляет запись `<xxx, value>` в таблицу символов, где *value* — это следующий доступный адрес в пространстве оперативной памяти, предназначенном для переменных, и 2) завершает трансляцию инструкции, используя этот адрес. В платформе Hack пространство ОЗУ, предназначенное для хранения переменных, начинается с 16 и увеличивается на 1 каждый раз, когда в коде встречается новая переменная. На рисунке 6.1 второй проход приводит к добавлению в таблицу символов `i` и `sum`.

6.4. Реализация

Использование. Ассемблер Hack принимает один аргумент командной строки, следующим образом:

```
prompt> HackAssembler Prog.asm,
```

где входной файл *Prog.asm* содержит ассемблерные команды (расширение `.asm` обязательно). Имя файла может содержать путь к файлу. Если путь не указан, ассемблер работает с текущей папкой. Ассемблер создает выходной файл с именем *Prog.hack* и записывает в него переведенные двоичные инструкции. Выходной файл создается в той же папке, что и входной. Если в папке есть файл с таким именем, то он будет перезаписан.

Мы предлагаем разделить реализацию ассемблера на два этапа. На первом этапе разработать базовый ассемблер для программ Hack, не содержащих символьных ссылок. На втором этапе расширить базовый ассемблер для работы с символьными ссылками.

6.4.1. Разработка базового ассемблера

Базовый ассемблер предполагает, что исходный код не содержит символьных ссылок. Поэтому, за исключением обработки комментариев и пробелов, ассемблер должен переводить либо C-команды, либо A-команды вида `@xxx`, где `xxx` — десятичное значение (а не символ).

Эта задача перевода проста: каждый мнемонический компонент (поле) символической *C*-команды переводится в соответствующий бит кода, согласно иллюстрации 6.2, а каждая десятичная константа *xxx A*-кода переводится в соответствующий двоичный вид.

Мы предлагаем построить ассемблер на основе программной архитектуры, состоящей из модуля *Parser* для разбора входных данных на команды и команд на поля, модуля *Code* для перевода полей (символьных мнемоник) в двоичные коды и программы *Hack assembler*, которая управляет всем процессом перевода. Прежде чем перейти к описанию этих трех модулей, мы хотим сделать замечание о стиле, который используем для описания этих спецификаций.

Документация API: разработка ассемблера Hack — первый из серии семи проектов по созданию программного обеспечения, которым мы будем заниматься во второй части книги. Каждый из этих проектов можно разрабатывать независимо, с использованием любого языка программирования высокого уровня. Поэтому в нашем стиле документации API нет никаких предложений относительно языка реализации.

В каждом проекте, начиная с этого, мы предлагаем API, состоящий из нескольких *модулей*. Каждый модуль документирует одну или несколько *процедур*. В типичном объектно-ориентированном языке модуль соответствует *классу*, а процедура (операция модуля) — *методу*. В других языках модуль может соответствовать *файлу*, а процедура — *функции*. Какой бы язык вы ни использовали для реализации программных проектов, начиная с ассемблера, не должно возникнуть проблем с воплощением модулей и процедур предлагаемых нами API в программные элементы выбранного вами языка реализации.

Модуль Parser

Модуль Parser («парсер») воплощает собой доступ к входному ассемблерному коду. В частности, он обеспечивает удобное средство для продвижения по исходному коду с пропуском комментариев и пробелов и разбиением каждой символической команды на ее базовые компоненты. Хотя базовая версия ассемблера не обязана обрабатывать

символьные ссылки, парсер, который мы определяем ниже, делает это. Другими словами, указанный здесь парсер обслуживает как базовый ассемблер, так и полный ассемблер.

Парсер игнорирует комментарии, пробелы в начале строк и пропуска строки во входном потоке, вводит строки по одной за раз и разбирает символьные команды на их базовые компоненты.

API модуля Parser изложен на следующей странице. Приведем несколько примеров использования функций парсера. Если текущая команда @17 или @sum, вызов `symbol()` вернет строку «17» или «sum» соответственно. Если текущая команда (LOOP), то вызов `symbol()` вернет строку «LOOP». При текущей команде `D = D + 1;` `JLE`, вызов `dest()`, `comp()` и `jump()` вернет строки «D», «D + 1» и «JLE» соответственно.

В проекте 6 вы должны реализовать этот API с помощью какого-либо языка программирования высокого уровня. Для этого вы должны знать, как данный язык работает с текстовыми файлами и строками.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Файл ввода/ поток	—	Открывает входной файл / поток и подготавливает его к разбору
hasMoreLines	—	Булево значение (boolean)	Есть ли еще строки на входе?
advance	—	—	При необходимости пропускает пробелы и разрывы. Читает со входа следующую команду и делает ее текущей. Эту рутину следует вызывать, только если hasMoreLines равно true. Изначально текущей команды нет
instructionType	—	A_INSTRUCTION, C_INSTRUCTION, L_INSTRUCTION (константы)	Возвращает тип текущей команды: A_INSTRUCTION для @xxx, где xxx — либо десятичное число, либо символ. C_INSTRUCTION для <i>dest=comp;</i> <i>jump</i> . L_INSTRUCTION для (xxx), где xxx — символ

Процедура	Аргументы	Возвращает	Функция
symbol	—	Строку	Если текущая команда (<i>xxx</i>), возвращает символ <i>xxx</i> . Если текущая команда <i>@xxx</i> , возвращает символ или десятичное значение <i>xxx</i> (в виде строки). Следует вызывать, только если <code>instructionType</code> равен <code>A_INSTRUCTION</code> или <code>L_INSTRUCTION</code>
dest	—	Строку	Возвращает символическую часть <i>dest</i> текущей C-команды (8 вариантов). Следует вызывать, только если <code>instructionType</code> равен <code>C_INSTRUCTION</code>
comp	—	Строку	Возвращает символическую часть <i>comp</i> текущей C-команды (28 вариантов). Следует вызывать, только если <code>instructionType</code> равен <code>C_INSTRUCTION</code>
jump	—	Строку	Возвращает символическую часть <i>jump</i> текущей C-команды (8 вариантов). Следует вызывать, только если <code>instructionType</code> равен <code>C_INSTRUCTION</code>

Модуль Code

Этот модуль переводит символьные мнемоники Nask в их двоичные коды в соответствии со спецификациями языка (см. иллюстрацию 6.2). Вот его API.

Процедура	Аргументы	Возвращает	Функция
dest	Строка	3 бита, в виде строки	Возвращает двоичный код мнемоники <i>dest</i>
comp	Строка	7 битов, в виде строки	Возвращает двоичный код мнемоники <i>comp</i>
jump	Строка	3 бита, в виде строки	Возвращает двоичный код мнемоники <i>jump</i>

Все n -битные коды возвращаются в виде строк из символов «0» и «1». Например, вызов `dest("DM")` возвращает строку «011», вызов `comp("A + 1")` возвращает строку «0110111», вызов `comp("M + 1")` возвращает строку «1110111», вызов `jump("JNE")` возвращает строку «101», и т. д. Все соотношения между мнемониками и двоичными кодами показаны на иллюстрации 6.2.

Модуль Assembler Hack

Это основная программа, которая управляет всем процессом сборки двоичной программы, пользуясь результатами работы модулей Parser и Code. Базовая версия ассемблера (которую мы сейчас описываем) предполагает, что исходный ассемблерный код не содержит символьных ссылок. Это означает, что: 1) во всех инструкциях типа `@xxx` константы `xxx` являются десятичными числами, а не символами, и 2) входной файл не содержит команд с метками, то есть инструкций вида `(xxx)`.

На этом этапе базовую программу ассемблера можно описать следующим образом. Программа получает имя входного исходного файла, скажем, *Prog*, из аргумента командной строки. Она запускает парсер для разбора входного файла *Prog.asm* и создает выходной файл *Prog.hack*, в который будет записывать переведенные двоичные команды. Затем программа входит в цикл, который итеративно просматривает строки (команды ассемблера) во входном файле и обрабатывает их, как описано ниже.

Для каждой *C*-команды программа использует сервисы Parser и Code для разбора команды на поля и перевода каждого поля в соответствующий двоичный код. Затем программа собирает (конкатенирует) переведенные двоичные коды в строку, состоящую из шестнадцати символов «0» и «1», и записывает эту строку в качестве следующей строки в выходной файл *.hack*.

Для каждой *A*-команды типа `@xxx` программа переводит число `xxx` в его двоичное представление, создает строку из шестнадцати символов «0» и «1» и записывает ее в качестве следующей строки в выходной файл *.hack*.

API для этого модуля мы не предоставляем, предлагая вам реализовать его по своему усмотрению.

6.4.2. Завершение создания ассемблера

Таблица символов

Поскольку команды Nask могут содержать символические ссылки, ассемблер должен преобразовывать их в фактические адреса. Решает эту задачу он с помощью таблицы символов, предназначенной для определения и хранения соответствий между символами и их значениями (в случае с Nask — адресами ОЗУ и ПЗУ).

Естественным средством для реализации этого отображения <символ, адрес> будет любая структура данных, предназначенная для работы с парами <ключ, значение>. Каждый современный высокоуровневый язык программирования имеет такую готовую абстракцию, обычно называемую *хеш-таблицей*, *картой*, *словарем* и т. д. Вы можете либо сами реализовать таблицу символов с нуля, либо настроить одну из этих структур данных. Вот API таблицы символов.

Процедура	Аргумент	Возвращает	Функция
Конструктор/инициализатор	—	—	Создает новую пустую таблицу символов
addEntry	symbol (строка), address (целое число, int)	—	Добавляет в таблицу <symbol, address>
contains	symbol (строка)	Булево значение (boolean)	Содержит ли таблица символов данный символ symbol?
getAddress	symbol (строка)	Целое число (int)	Возвращает адрес address, ассоциируемый с символом symbol

6.5 Проект

Задача: разработать ассемблер, который переводит программы, написанные на языке ассемблера Hack, в двоичный код Hack.

Эта версия ассемблера предполагает, что исходный код ассемблера не содержит ошибок. Проверку на наличие ошибок и обработку ошибок можно будет добавить в последующие версии ассемблера, но в проект 6 они не входят.

Ресурсы: основной инструмент, необходимый для выполнения данного проекта, — это язык программирования, на котором вы будете реализовывать свой ассемблер. Также могут пригодиться поставляемые ассемблер и эмулятор ЦПУ, расположенные в папке `nand2tetris/tools`. Эти инструменты позволяют поэкспериментировать с работающим ассемблером, прежде чем приступить к его созданию самостоятельно. Важно отметить, что поставляемый ассемблер позволяет сравнивать его вывод с выводами, сгенерированными вашим ассемблером. Для получения дополнительной информации об этих возможностях обратитесь к учебнику по ассемблеру на сайте www.nand2tetris.org.

Контракт: при запуске вашего ассемблера в командной строке с аргументом имени файла `Prog.asm`, содержащего корректную программу на языке ассемблера Hack, этот файл должен переводиться в корректный двоичный код Hack и сохраняться в файле с именем `Prog.hack` в той же папке, что и исходный файл (если файл с таким именем существует, он перезаписывается). Код, созданный вашим ассемблером, должен быть идентичен коду, производимому поставляемым ассемблером.

План разработки: мы предлагаем создавать и тестировать ассемблер в два этапа. Сначала напишите базовый ассемблер, предназначенный для трансляции программ, не содержащих символьных ссылок. Затем расширьте свой ассемблер возможностью работы с символами.

Тестовые программы: первая тестовая программа не имеет символических ссылок. Остальные тестовые программы представлены в двух версиях: *Prog.asm* и *ProgL.asm*, соответственно с символическими ссылками и без них.

Add.asm: складывает константы 2 и 3 и помещает результат в R0.

Max.asm: вычисляет $\max(R0, R1)$ и помещает результат в R2.

Rect.asm: рисует прямоугольник в левом верхнем углу экрана. Прямоугольник имеет в ширину 16 пикселей и в высоту R0 пикселей. Перед запуском этой программы поместите в R0 неотрицательное значение.

Pong.asm: классическая аркадная игра для одного игрока. По экрану движется мячик, отскакивая от краев экрана. Игрок пытается отбить мяч ракеткой, перемещая ракетку с помощью клавиш со стрелками влево и вправо. За каждое удачное попадание игрок получает очко, а ракетка немного уменьшается, чтобы усложнить игру. Если игрок промахивается по мячу, игра заканчивается. Чтобы выйти из игры, нажмите ESC.

Поставляемая программа «Pong» была разработана с использованием инструментов, описанных во второй части книги. В частности, игровая программа была написана на языке программирования высокого уровня Jack и переведена в данный файл *Pong.asm* компилятором *Jack*. Хотя высокоуровневая программа *Pong.jack* состоит всего из трехсот строк кода, исполняемое приложение «Pong» содержит около двадцати тысяч строк двоичного кода, большую часть которого составляет операционная система Jack. В прилагаемом эмуляторе процессора программа работает медленно, так что не ждите от игры «Pong» особой скорости. Но ее медлительность в данном случае даже идет на пользу, поскольку позволяет отслеживать графическое поведение программы. После того как во второй части вы разработаете иерархию программного обеспечения, игра будет работать намного быстрее.

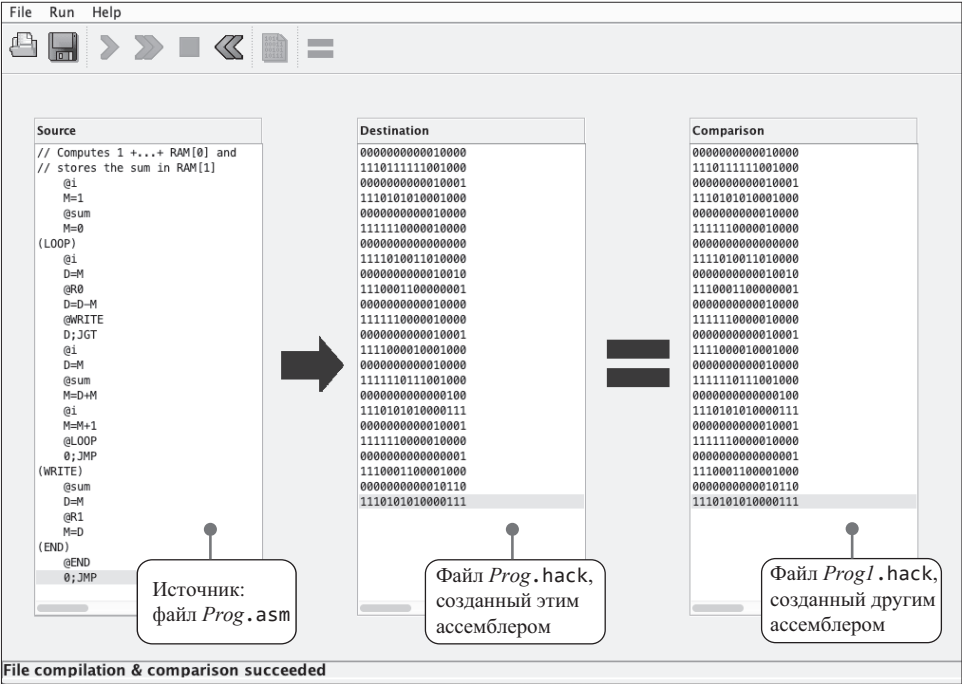


Иллюстрация 6.3. Тестирование ассемблера с помощью поставляемого ассемблера.

Тестирование: предположим, что у нас для тестирования имеется программа *Prog.asm* — одна из программ, написанных на языке ассемблера Nask. Проверить правильность работы вашего ассемблера можно двумя способами. Во-первых, вы можете загрузить файл *Prog.hack*, сгенерированный вашим ассемблером, в прилагаемый эмулятор процессора, выполнить его и проверить, действительно ли он делает то, что должен делать.

Второй способ тестирования заключается в сравнении кода, сгенерированного вашим ассемблером, с кодом, сгенерированным прилагаемым ассемблером. Для начала переименуйте файл, сгенерированный вашим ассемблером, в *Prog1.hack*. Затем загрузите *Prog.asm* в поставляемый ассемблер и транслируйте его в двоичный код. Если ваш ассемблер работает правильно, то файл *Prog1.hack* будет идентичен файлу *Prog.hack*, созданному поставляемым ассемблером.

Сравнение можно выполнить, загрузив *Prog1.asm* в качестве файла сравнения — подробности см. на иллюстрации 6.3.

Веб-версия проекта 6 доступна на сайте www.nand2tetris.org.

6.6. Перспектива

Как и большинство ассемблеров, ассемблер Hack — это относительно простой «транслятор», занимающийся в основном обработкой текста. Естественно, ассемблеры для более богатых машинных языков тоже более сложны. Кроме того, некоторые ассемблеры обладают широкими возможностями работы с символами, которых нет в Hack. К примеру, некоторые ассемблеры поддерживают *арифметические операции с константами и символами*, например, выражение `base + 5` может служить ссылкой на пятую ячейку памяти после адреса, на который ссылается `base`.

Многие ассемблеры могут обрабатывать макрокоманды. Макрокоманды — это последовательность машинных команд, имеющая свое отдельное имя. Например, наш ассемблер можно расширить для перевода заранее установленных макрокоманд, например `D = M[addr]`, в две примитивные Hack-команды: `@addr`, за которой следует `D = M`. Аналогично можно придумать макрокоманду `goto addr`, которая будет транслироваться как команда `@addr`, за которой следует `0`; `JMP`, и т. д. Такие макрокоманды значительно упрощают написание программ на ассемблере при низких затратах на трансляцию.

Следует отметить, что программы на машинном языке редко пишутся людьми. Они, как правило, составляются компиляторами. А компилятор, будучи автоматом, может обходиться и вовсе без символьных команд и генерировать двоичный машинный код напрямую. Тем не менее ассемблер остается полезной программой, особенно для разработчиков программ на C/C++, которые заботятся об эффективности и оптимизации. Проверяя символьный код, сгенерированный компилятором, программист может улучшить высокоуровневый код и добиться лучшей производительности на выбранном аппаратном

обеспечении. Когда же сгенерированный ассемблерный код будет признан эффективным, ассемблер переведет его в конечный двоичный исполняемый код.

Поздравляем! Вы достигли конца первой части путешествия «От Nand до “Тетриса”». Если вы выполнили проекты 1–6, то построили компьютерную систему общего назначения на основе основополагающих базовых принципов. Это фантастическое достижение, и вы должны чувствовать гордость и удовлетворение.

К сожалению, наш компьютер способен выполнять только программы, написанные на машинном языке, так что во второй части книги мы воспользуемся этой «голой» аппаратной платформой в качестве отправной точки и построим на ее основе современную иерархию программного обеспечения. Программное обеспечение будет состоять из виртуальной машины, компилятора для высокоуровневого объектно-ориентированного языка программирования и базовой операционной системы.

Итак, если вы готовы к новым приключениям, перейдем ко второй части нашего грандиозного путешествия «От Nand до “Тетриса”».

II. Программное обеспечение

Любая достаточно развитая технология неотличима от магии.

— Артур Кларк

К этому добавим: «И любая достаточно развитая магия неотличима от тяжелой закулисной работы». В первой части книги мы построили аппаратную платформу компьютерной системы под названием Наск, способной выполнять программы, написанные на машинном языке Наск. Во второй части превратим эту «голую» машину в передовую технологию, неотличимую от магии: «черный ящик», который может превратиться в шахматиста, поисковую систему, симулятор полета, медиацентр или во что угодно, что придет вам в голову. А для этого нужно раскрыть сложную закулисную иерархию программного обеспечения, которая наделяет компьютеры способностью выполнять программы, написанные на языках программирования высокого уровня. В частности, мы сосредоточимся на Jack — простом и похожем на Java объектно-ориентированном языке программирования, формально описанном в главе 9. На протяжении многих лет читатели нашей книги и студенты практического курса «От Nand до “Тетриса”» использовали язык Jack для разработки множества игр, в том числе «Тетриса», «Pong», «Змейки», «Space Invaders», а также множества других интерактивных приложений. Будучи компьютером общего назначения, Наск может выполнять все эти программы, а также любые другие, которые вы напишете по своему желанию.

Очевидно, что разрыв между выразительным синтаксисом высокоуровневых языков программирования и неуклюжими командами-инструкциями низкоуровневого машинного языка огромен. Если вы хотите поспорить с этим, попробуйте разработать игру «Тетрис» с помощью только таких команд, как @17 и $M = M + 1$. Преодоление этого разрыва — вот чему посвящена вторая часть этой книги. Мы построим этот мост, постепенно разрабатывая некоторые из самых мощных и важнейших программ в прикладной информатике: компилятор, виртуальную машину и базовую операционную систему.

Наш компилятор Jack будет предназначаться для того, чтобы взять какую-нибудь программу на языке Jack, скажем, «Тетрис», и создать из нее список команд машинного языка, который при своем воспроизведении заставит платформу Nask воспроизвести данную игру. Конечно, «Тетрис» — это только один пример: компилятор, который вы создадите, сумеет перевести любую заданную программу Jack в машинный код, который может быть выполнен на компьютере Nask. Компилятор, основные задачи которого заключаются в синтаксическом анализе и генерации кода, будет построен в главах 10 и 11.

Как и для языков программирования вроде Java и C#, компилятор Jack будет *двухуровневым*: он будет генерировать промежуточный *ВМ-код*, предназначенный для запуска на абстрактной *виртуальной машине* (ВМ). Затем ВМ-код будет компилироваться отдельным транслятором в машинный язык Nask. Виртуализация — одна из самых важных идей в прикладной информатике — используется во многих областях, включая компиляцию программ, облачные вычисления, распределенное хранение данных, распределенную обработку и операционные системы. Мы посвятим главы 7 и 8 целям, проектированию и созданию нашей виртуальной машины.

Как и многие другие высокоуровневые языки, базовый язык Jack удивительно прост. В мощные системы программирования современные языки превращаются благодаря *стандартным библиотекам*, предоставляющим математические функции, обработку строк, управление памятью, отображение графики, обработку взаимодействия с пользователем и многое другое. Взятые вместе, эти стандартные библиотеки образуют базовую *операционную систему* (ОС), которая

в рамках Jack упакована как *стандартная библиотека классов Jack*. Эта базовая ОС, призванная устранить многие разрывы между языком высокого уровня Jack и низкоуровневой платформой Nask, будет разработана средствами самого языка Jack. Вы, должно быть, недоумеваете, как это программное обеспечение, которое должно обеспечивать работу языка программирования, может разрабатываться на самом языке программирования. Мы решим данную проблему, следуя стратегии разработки под названием «бутстрэппинг», или «раскрутка компилятора», похожей на тот метод, с помощью которого ОС Unix была разработана с использованием языка C.

Работа над созданием ОС послужит нам возможностью описать элегантные алгоритмы и классические структуры данных, которые обычно используются для управления аппаратными ресурсами и периферийными устройствами. Затем мы реализуем эти алгоритмы на языке Jack, шаг за шагом расширяя возможности языка. По мере изучения глав второй части вы будете рассматривать ОС с нескольких различных точек зрения. В главе 9, выступая в роли *прикладного программиста*, вы разработаете приложение на языке Jack и будете использовать службы ОС абстрактно, с точки зрения клиента высокого уровня. В главах 10 и 11 при создании компилятора Jack будете использовать службы ОС как низкоуровневый клиент, например, различные службы управления памятью, необходимые компилятору. И, наконец, в главе 12 вы сядете в кресло разработчика ОС и самостоятельно реализуете все эти системные службы.

II.1. Примеры программирования на языке Jack

Прежде чем приступить к изучению всех этих захватывающих проектов, мы постараемся дать вам краткое и неформальное представление о языке Jack. Сделаем мы это на двух примерах, начиная с Hello World. Данный пример докажет, что даже самая тривиальная программа высокого уровня скрывает под собой гораздо больше, чем кажется на первый взгляд. Затем покажем простую программу, демонстрирующую объектно-ориентированные возможности языка Jack.

Почувствовав, так сказать, «вкус» высокоуровневого языка Jack, ориентированного на программистов, мы будем готовы начать путь к реализации языка посредством создания виртуальной машины, компилятора и операционной системы.

И снова Hello World: мы начали эту книгу с культовой программы по выведению на экран надписи «*Hello World*», с которой часто сталкиваются учащиеся на вводных курсах по программированию. Приведем эту тривиальную программу, написанную на языке программирования Jack, еще раз:

```
// Первый пример программирования 101
class Main {
    function void main () {
        do Output.printString ("Hello World");
        return;
    }
}
```

Обсудим некоторые неявные предположения, которые мы обычно делаем, когда нам показывают такие программы. Первое «волшебство», которое мы воспринимаем как должное, заключается в том, что последовательность символов, скажем, `printString ("Hello World")`, действительно заставляет компьютер что-то вывести на экран. Но как компьютер узнает, *что* ему делать? И даже если он знает, что делать, то *как* он это делает? Как было показано в первой части книги, экран представляет собой сетку пикселей. Если мы хотим отобразить на экране букву Н, мы должны включать и выключать специально отобранные подмножества пикселей, которые, взятые вместе, создают на экране изображение нужной буквы. Конечно, это только самое начало. Как, например, отобразить букву Н на экранах разного размера и разрешения? А как быть с циклами `while` и `for`, массивами, объектами, методами, классами и прочими возможностями и условностями языков высокого уровня, которыми программисты обучены пользоваться, не задумываясь о том, как они работают?

И в самом деле, вся прелесть языков программирования высокого уровня, да и вообще хорошо продуманных абстракций, заключается в том, что мы можем пользоваться всеми их преимуществами, оставаясь в состоянии «блаженного неведения». Программистам практически предлагают рассматривать язык как некий абстрактный «черный ящик» без уделения внимания его реализации. Достаточно хорошего учебника, нескольких примеров кода — и вот вы уже начинаете работать.

Однако очевидно, что в тот или иной момент *кто-то* должен был реализовать эту языковую абстракцию. Кто-то должен был раз и навсегда разработать метод эффективного вычисления квадратных корней, который теперь любой прикладной программист вызывает, вставляя, например, в свою программу команду `sqrt(1764)`. Кто-то должен был придумать, как вводить в память набираемое пользователем число, когда программист пишет `x = readInt()`, и кто-то же придумал, как находить и использовать свободный блок памяти, когда программист парой нажатий клавиш создает объект `new`. Все это касается и всех других абстрактных сервисов, которые программисты ожидают получить, никогда не задумываясь об их реализации. Итак, кто же те добрые души, превращающие высокоуровневое программирование в передовую технологию, неотличимую от магии? Это как раз те волшебники из области разработки программного обеспечения, которые разрабатывают *компиляторы, виртуальные машины и операционные системы*. Именно этим вы и будете заниматься в последующих главах.

Вы можете задаться вопросом, зачем вам вообще знать о том, что происходит где-то на заднем плане, за кулисами сцены. Разве мы только что не сказали, что высокоуровневыми языками можно пользоваться, не беспокоясь о том, как они работают? На это есть как минимум две причины. Во-первых, чем больше вы углубляетесь в низкоуровневую систему, тем более искусным программистом становитесь. В частности, вы учитесь писать высокоуровневый код, эффективно использующий аппаратное обеспечение и ОС, и избегать таких неприятных ошибок, как утечки памяти.

Во-вторых, закатав рукава и самостоятельно разработав внутренние компоненты системы, вы откроете для себя некоторые из самых

красивых и мощных алгоритмов и структур данных в прикладной информатике. Важно отметить, что обсуждаемые во второй части книги идеи и методы не ограничиваются компиляторами и операционными системами. Это скорее строительные блоки многочисленных программных систем и приложений, которые будут сопровождать вас на протяжении всей карьеры.

Программа PointDemo: предположим, мы хотим отображать точки на плоскости и манипулировать ими. На иллюстрации II.1 показаны две такие точки, p_1 и p_2 , и третья точка p_3 , полученная в результате сложения векторов $p_3 = p_1 + p_2 = (1, 2) + (3, 4) = (4, 6)$. На иллюстрации также показано *евклидово расстояние* между p_1 и p_3 , которое можно вычислить с помощью теоремы Пифагора. Код в классе Main показывает, как такие алгебраические манипуляции выполняются с помощью объектно-ориентированного языка Jack.

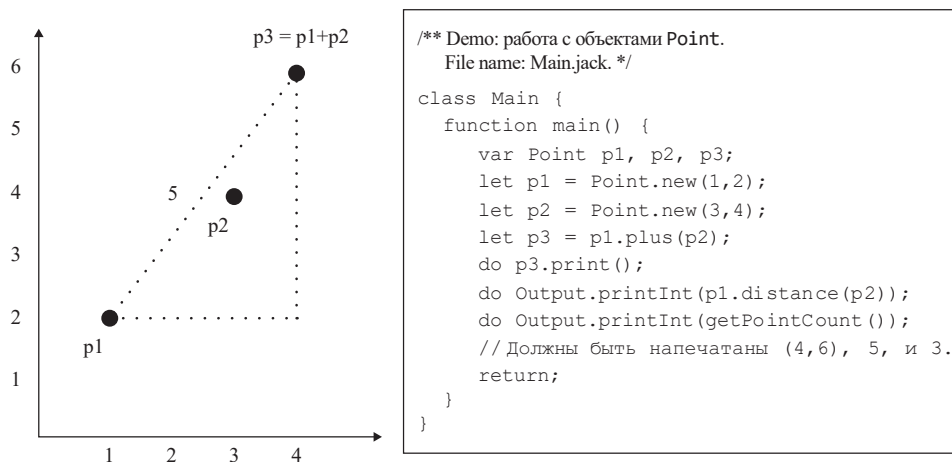


Иллюстрация II.1. Манипуляция точками на плоскости: пример и код на языке Jack.

Вы можете задаться вопросом, зачем в Jack используются такие ключевые слова, как `var`, `let` и `do`. Пока что мы советуем вам не заикливаться на синтаксических подробностях. Сосредоточимся вместо

этого на общей картине и рассмотрим, как на языке Jack можно реализовать абстрактный тип данных Point (иллюстрация II.2).

Код, показанный на иллюстрации II.2, говорит о том, что класс Jack (примерами которого являются Main и Point) представляет собой набор одной или нескольких *процедур (подпрограмм)*, каждая из которых — это *конструктор, метод* или *функция*. Конструкторы — это процедуры, создающие новые объекты; *методы* — процедуры, работающие с текущим объектом, а *функции* — процедуры, не работающие с каким-то конкретным объектом. Приверженцы объектно-ориентированного дизайна могут нахмуриться: как это мы в одном классе смешали методы и функции; мы делаем это здесь для примера.

```

/** Представляет точки на двумерной плоскости.
    File name: Point.jack. */
class Point {
    // Координаты этой точки:
    field int x, y
    // Количество до сих пор созданных объектов Point:
    static int pointCount;
    /** Создает точку на двумерной плоскости
        и инициализирует ее с заданными координатами. */
    constructor Point new(int ax, int ay) {
        let x \ ax;
        let y \ ay;
        let pointCount \ pointCount + 1;
        return this;
    }
    /** Возвращает координату x этой точки. */
    method int getX() {return x;}
    /** Возвращает координату y этой точки. */
    method int getY() {return y;}
    /** Возвращает количество до сих пор созданных
        объектов Point. */
    function int getPointCount() {
        return pointCount;
    }
    // Объявление класса продолжается сверху справа.

```

```

    /** Возвращает точку, являющуюся суммой
        этой и другой. */
    method Point plus(Point other) {
        return Point.new(x + other.getX(),
            y + other.getY());
    }
    /** Возвращает евклидово расстояние между
        этой точкой и другой. */
    method int distance(Point other) {
        var int dx, dy;
        let dx \ x - other.getX();
        let dy \ y - other.getY();
        return Math.sqrt((dx*dx) + (dy*dy));
    }
    /** Печатает эту точку как "(x,y)". */
    method void print() {
        do Output.printString("(");
        do Output.printInt(x);
        do Output.printString(",");
        do Output.printInt(y);
        do Output.printString(")");
        return;
    }
} // Конец объявления класса Point.

```

Иллюстрация II.2. Реализация абстракции Point на языке Jack.

В оставшейся части этого раздела предложен неформальный обзор классов Main и Point. Наша цель — дать представление о программировании на языке Jack, отложив полное описание языка до главы 9. Итак, позволим себе роскошь сосредоточиться только на сути.

Функция `Main.main` начинается с объявления трех *объектных переменных* (также называемых *ссылками*, или *указателями*), предназначенных для ссылок на экземпляры класса `Point`. Затем она создает два объекта `Point` и присваивает им переменные `p1` и `p2`. После этого вызывается метод `plus` и присваивает переменную `p3` объекту `Point`, возвращенному этим методом. Остальная часть функции `Main.main` выводит на печать некоторые результаты.

Класс `Point` начинается с объявления того, что каждый объект `Point` характеризуется двумя *переменными поля* (также называемыми *свойствами*, или *переменными экземпляра*). Затем объявляется *статическая переменная*, то есть переменная уровня класса, не связанная ни с каким конкретным объектом. Конструктор класса устанавливает значения полей вновь создаваемого объекта и увеличивает количество экземпляров этого класса, созданных на данный момент. Обратите внимание, что конструктор `Jack` должен явно возвращать адрес памяти нового создаваемого объекта, который, согласно правилам языка, обозначается `this`.

Вы можете удивиться, почему результат вычисления квадратного корня методом `distance` хранится в переменной типа `int` — очевидно, что более логичным было бы использование вещественного типа данных, например, `float`. Причина этой особенности проста: в языке `Jack` имеются только три примитивных типа данных: `int`, `boolean` и `char`. Другие типы данных по желанию можно реализовать с помощью классов, как мы это сделаем в главах 9 и 12.

Операционная система: классы `Main` и `Point` используют три функции ОС: `Output.printInt`, `Output.printString` и `Math.sqrt`. Как и другие современные языки высокого уровня, язык `Jack` дополнен набором *стандартных классов*, предоставляющих часто используемые службы ОС (полный API ОС приведен в приложении 6). Мы еще многое расскажем о сервисах ОС в главе 9, где будем использовать их в контексте программирования на языке `Jack`, а также в главе 12, где мы создадим ОС.

Помимо вызова сервисов ОС непосредственно из программ `Jack`, операционная система задействуется и другими менее очевидными

способами. Рассмотрим, например, операцию `new`, используемую для построения объектов в объектно-ориентированных языках. Как компилятор узнает, куда в оперативную память машины RAM поместить только что созданный объект? Никак. Чтобы выяснить это, вызывается особая процедура ОС. Когда мы будем создавать ОС в главе 12, вы, помимо всего другого, реализуете и типичную систему управления памятью во время выполнения. Затем вы на практике узнаете, как эта система взаимодействует с аппаратным обеспечением с одной стороны и с компиляторами с другой, чтобы умно и эффективно выделять и освобождать пространство оперативной памяти. Это лишь один пример, иллюстрирующий, как ОС устраняет разрыв между высокоуровневыми приложениями и аппаратной платформой хоста.

II.2. Компиляция программы

Программа высокого уровня — это символическая абстракция, ничего не значащая для машинного оборудования. Перед выполнением программы высокоуровневый код нужно перевести на машинный язык. Такой процесс перевода называется *компиляцией*, а выполняющая его программа называется *компилятором*. Написание компилятора, переводящего высокоуровневые программы в низкоуровневые машинные инструкции, — весьма достойная задача. Некоторые языки, например Java и C#, решают эту задачу с помощью элегантной *двухуровневой* модели компиляции. Сначала исходная программа переводится в промежуточный абстрактный код виртуальной машины (называемый *байт-кодом* в Java и Python и *промежуточным языком* в C#/.NET). Затем, в ходе совершенно отдельного и независимого процесса, ВМ-код можно перевести на машинный язык любой целевой аппаратной платформы.

Эта модульность — по крайней мере, одна из причин, почему Java стал таким доминирующим языком программирования. В исторической перспективе Java можно рассматривать как мощный объектно-ориентированный язык, чья двухуровневая модель компиляции оказалась полезной как раз в нужное время, когда компьютеры начали превращаться из нескольких предсказуемых платформ процессоров/

ОС в запутанную смесь многочисленных ПК, сотовых телефонов, мобильных устройств и устройств Интернета вещей, объединенных в глобальную сеть. Написать высокоуровневую программу, которая будет выполняться на любой из этих платформ, — невероятно сложная задача. Один из способов упорядочить (с точки зрения компиляции) такую пеструю экосистему, созданную разными производителями, — это свести ее к некоей общей согласованной архитектуре виртуальных машин. Выступая в качестве общей, промежуточной среды выполнения, виртуальные машины позволяют разработчикам писать высокоуровневые программы, работающие почти «как есть» на многих различных аппаратных платформах, каждая из которых имеет собственную реализацию ВМ. Во второй части мы еще не раз будем рассуждать о возможностях модульного подхода.

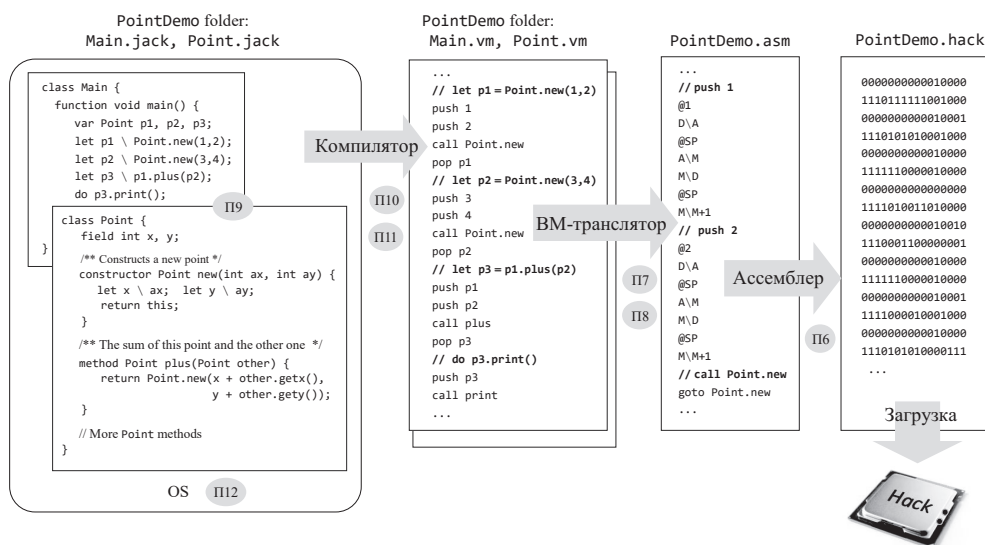


Иллюстрация II.3. Дорожная карта второй части книги (ассемблер относится к первой части и показан здесь для полноты картины). Дорожная карта описывает иерархию трансляции: от высокоуровневой, объектно-ориентированной, многоклассовой программы к коду ВМ, от кода ВМ к коду на языке ассемблера и от кода на языке ассемблера к исполняемому двоичному коду. Пронумерованные кружки обозначают проекты по реализации компилятора, транслятора ВМ, ассемблера и операционной системы. Проект 9 посвящен написанию приложения на языке Jack для ознакомления с этим языком.

Путь, который вас ожидает: в оставшихся главах книги мы займемся разработкой всех упомянутых выше интересных программных технологий. Наша конечная цель — создание инфраструктуры для превращения высокоуровневых программ — *любых* программ — в исполняемый код. Дорожная карта показана на иллюстрации II.3.

Следуя принципу «От Nand до “Тетриса”», мы будем двигаться по дорожной карте второй части снизу вверх. Для начала предположим, что у нас есть аппаратная платформа, оснащенная языком ассемблера. В главах 7–8 мы опишем архитектуру виртуальной машины и язык VM, а также реализуем абстракцию, разработав транслятор VM, переводящий программы VM в программы на ассемблере Nask. В главе 9 опишем язык высокого уровня Jack и используем его для разработки простой компьютерной игры. Таким образом, вы познакомитесь с языком и операционной системой Jack до того, как приступите к их созданию. В главах 10–11 мы разработаем компилятор Jack, а в главе 12 создадим операционную систему.

Итак, засучим рукава и приступим к работе!

7. Виртуальная машина I: обработка

Программисты — это создатели вселенных, за которые отвечают только они. В виде компьютерных программ могут быть созданы вселенные практически неограниченной сложности.

— Джозеф Вейценбаум, «Возможности
вычислительных машин и человеческий разум» (1974)

В этой главе описываются первые шаги по созданию компилятора для типичного объектно-ориентированного языка высокого уровня. Мы подходим к решению этой задачи в два основных этапа, каждый из которых занимает две главы. В главах 10–11 мы опишем создание *компилятора*, предназначенного для перевода высокоуровневых программ в *промежуточный код*; в главах 7–8 обрисует создание последующего *транслятора*, предназначенного для перевода промежуточного кода на машинный язык целевой аппаратной платформы. Как следует из номеров глав, в своих рассуждениях и описаниях мы будем руководствоваться направлением «снизу вверх», начиная с транслятора.

Промежуточный код, лежащий в основе данной модели компиляции, предназначен для выполнения на абстрактном компьютере, называемом *виртуальной машиной*, или ВМ. Есть несколько причин, по которым эта двухуровневая модель компиляции имеет смысл по сравнению с традиционными компиляторами, переводящими высокоуровневые программы непосредственно на машинный язык. Одно из преимуществ — кроссплатформенная совместимость: поскольку виртуальную машину относительно легко реализовать на различных

аппаратных платформах, один и тот же код ВМ в исходном виде может работать на любом устройстве, оснащенном такой же реализацией ВМ. И это одна из причин, по которой Java стал доминирующим языком в сфере разработки приложений для мобильных устройств, для которой характерно огромное множество самых разных комбинаций процессоров и оперативных систем. ВМ можно реализовать на целевых устройствах с помощью программных интерпретаторов, специализированного оборудования или посредством перевода программ ВМ на машинный язык устройства. Последний подход к реализации используется для языков Java, Scala, C# и Python, а также для языка Jack, разработанного в рамках практического курса «От Nand до “Тетриса”».

В этой главе представлена типичная архитектура виртуальной машины и язык виртуальной машины, концептуально схожие с виртуальной машиной Java (JVM) и байт-кодом соответственно. Согласно общей концепции «От Nand до “Тетриса”» виртуальная машина будет представлена с двух точек зрения. Во-первых, мы объясним смысл виртуальной машины и опишем ее абстракцию, рассказав, для чего она предназначена. Затем опишем предлагаемую реализацию виртуальной машины на платформе Hack. Наша реализация предполагает написание программы под названием *ВМ-транслятор* (*VM translator*), переводящей код ВМ на ассемблерный код Hack.

Язык ВМ, который мы представим, состоит из арифметико-логических команд, команд доступа к памяти, называемых *push* и *pop*, команд ветвления и команд вызова и возврата функций. Мы разделили обсуждение и реализацию этого языка на две части, каждая из которых рассматривается в отдельной главе и проекте. В данной главе мы создаем базовый ВМ-транслятор, который реализует арифметико-логические команды ВМ и команды *push/pop*. В следующей главе расширим базовый транслятор для обработки команд ветвления и функций. В результате получим полномасштабную реализацию виртуальной машины, которая послужит основой для компилятора, который мы создадим в главах 10–11.

Виртуальная машина, которая появится в результате нашей работы, демонстрирует несколько важных идей и методов. Во-первых,

идея эмуляции одной вычислительной платформы средствами другой — это фундаментальная идея информатики, восходящая к Алану Тьюрингу и сформулированная еще в 1930-х годах. В настоящее время модель виртуальной машины лежит в основе нескольких распространенных сред программирования, включая Java, .NET и Python. Лучший способ получить глубокое представление о том, как работают эти среды программирования, — собрать простую версию ядра виртуальной машины, как мы это делаем здесь.

Еще одна важная тема данной главы — *стековая обработка*. *Стек* — это фундаментальная и элегантная структура данных, используемая в многочисленных компьютерных системах, алгоритмах и приложениях. Поскольку представленная здесь виртуальная машина основана на стеке, она представляет собой рабочий пример этой удивительно универсальной и мощной структуры данных.

7.1. Парадигма виртуальной машины

Для того чтобы программа, написанная на языке высокого уровня, запустилась на целевом компьютере, ее нужно перевести на машинный язык данного компьютера. Исторически для каждой пары языка высокого уровня и машинного языка низкого уровня разрабатывался отдельный компилятор. С годами увеличение количества языков высокого уровня с одной стороны и увеличение разнообразных типов процессоров и наборов инструкций с другой привели к распространению множества различных компиляторов, каждый из которых зависел от всех подробностей исходного и целевого языков. Один из способов устранить такую зависимость — разбить общий процесс компиляции на два почти отдельных этапа. На первом этапе высокоуровневый код разбирается и переводится (транслируется) в промежуточные и абстрактные шаги обработки, которые не являются ни высокоуровневыми, ни низкоуровневыми. На втором этапе промежуточные шаги переводятся на низкоуровневый машинный язык целевого оборудования.

Такая декомпозиция очень привлекательна с точки зрения программной инженерии. Сначала обратим внимание на то, что первый

этап трансляции зависит только от специфики исходного языка высокого уровня, а второй этап — только от специфики целевого низкоуровневого машинного языка. Конечно, для этого нужно тщательно разработать и оптимизировать интерфейс между двумя стадиями трансляции, то есть дать точное определение и спецификацию промежуточным шагам. На каком-то этапе эволюции трансляции программ разработчики компиляторов пришли к выводу, что этот промежуточный интерфейс достаточно важен, чтобы заслужить статус отдельного языка, предназначенного для работы на абстрактной машине. В частности, можно описать *виртуальную машину*, команды которой реализуют промежуточные шаги по обработке высокоуровневых команд и транслируют их. Компилятор, который раньше был единой монолитной программой, разделился теперь на две отдельные и гораздо более простые программы. Первая, по-прежнему называемая *компилятором*, переводит программу, написанную на языке высокого уровня, в промежуточные команды виртуальной машины; вторая программа, называемая *транслятором виртуальной машины*, переводит команды виртуальной машины в машинные инструкции целевой аппаратной платформы. На иллюстрации 7.1 показано, как эта двухуровневая система компиляции способствовала кроссплатформенной переносимости программ Java.

Система виртуальных машин включает в себе множество практических преимуществ. Когда производитель выводит на рынок новое цифровое устройство — допустим, мобильный телефон, — он относительно легко может разработать для него реализацию JVM, называемую средой выполнения для Java (Java Runtime Environment, JRE). Эта клиентская инфраструктура сразу же наделяет устройство огромной базой доступного программного обеспечения Java. А в мире .NET, где на один и тот же промежуточный язык виртуальной машины компилируются несколько языков высокого уровня, одна и та же виртуальная машина может быть совместима с компиляторами нескольких разных языков, что позволяет использовать общие программные библиотеки и обеспечивать совместимость этих языков между собой.

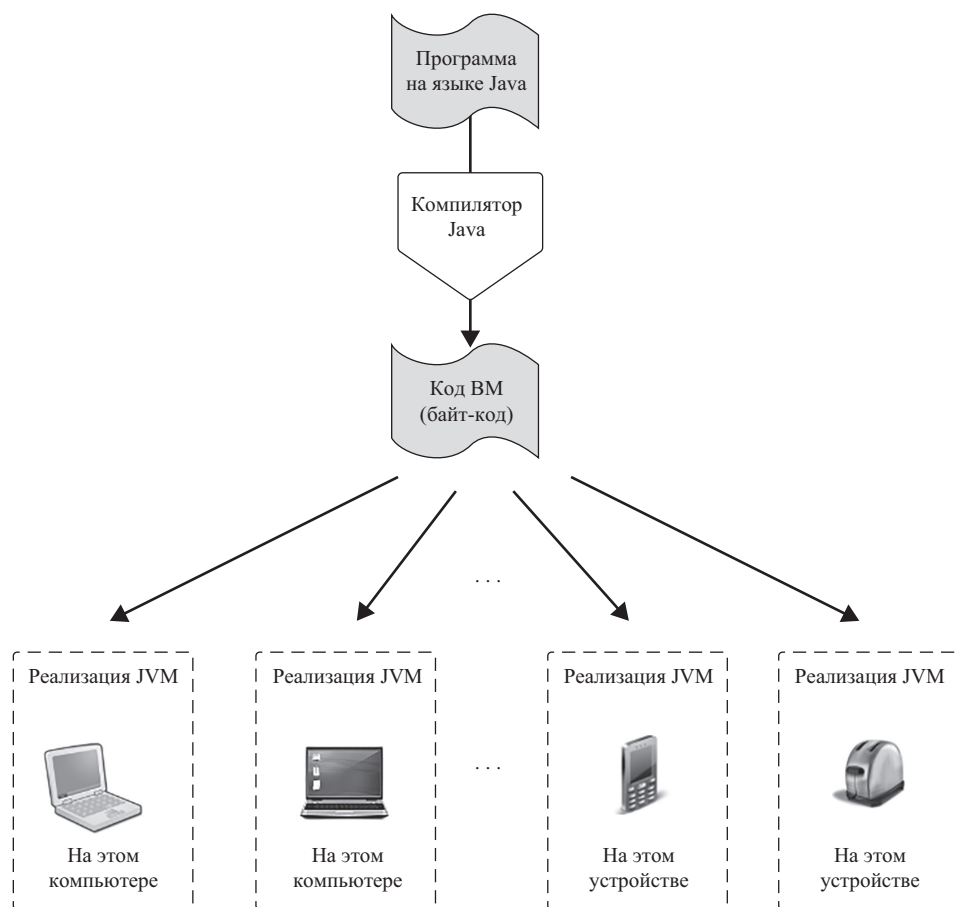


Иллюстрация 7.1. Структура виртуальной машины на примере Java. Высокоуровневые программы компилируются в промежуточный код виртуальной машины (ВМ). Один и тот же код ВМ можно отправить и выполнить на любой аппаратной платформе, оснащенной подходящей *реализацией виртуальной машины Java (JVM)*. Эти реализации виртуальной машины обычно представляют собой программы на стороне клиента, переводящие код ВМ на машинные языки целевых устройств.

Платой за элегантность и мощь виртуальной машины является снижение эффективности. Естественно, двухуровневый процесс трансляции в конечном счете приводит к генерации более многословного и громоздкого машинного кода по сравнению с машинным, который

можно получить при прямой компиляции. Однако по мере того как процессоры становятся все быстрее, а реализация ВМ все более оптимизированной, для большинства приложений снижение эффективности становится едва заметным. Конечно, всегда будут существовать высокопроизводительные приложения и встроенные системы, требующие эффективного кода, генерируемого одноуровневыми компиляторами таких языков, как С и С++. При этом современные версии С++ содержат как классические одноуровневые компиляторы, так и двухуровневые компиляторы на базе виртуальных машин.

7.2. Стековая машина

При разработке эффективного языка виртуальной машины необходимо найти удобный баланс между высокоуровневыми языками программирования с одной стороны и большим разнообразием низкоуровневых машинных языков с другой. Таким образом, желаемый ВМ-язык должен удовлетворять нескольким требованиям, идущим как «сверху», так и «снизу». Прежде всего язык должен обладать разумной выразительной силой. Ориентируясь на это требование, мы разработали язык ВМ, включающий в себя арифметико-логические команды, команды *push/pop*, команды ветвления и команды функций. Эти команды ВМ достаточно «высокие», чтобы код ВМ, генерируемый компилятором, был достаточно элегантным и хорошо структурированным. В то же время команды ВМ должны быть достаточно «низкими», чтобы машинный код, генерируемый на их основе ВМ-трансляторами, был плотным и эффективным. Говоря иначе, нужно было удостовериться в том, что разрывы в трансляции между высоким уровнем и уровнем ВМ с одной стороны и уровнем ВМ и машинным уровнем с другой не были слишком большими. Один из способов удовлетворить эти несколько противоречивые требования — создание промежуточного языка ВМ на основе специфической абстрактной архитектуры, называемой *стековой машиной*.

Прежде чем продолжить, хотели бы обратиться к вам с просьбой проявить терпение. Взаимосвязь между стековой машиной, которую

мы опишем далее, и компилятором, который мы представим позже в этой книге, не слишком очевидная. Мы советуем читателям позволить себе насладиться внутренней красотой абстракции стековой машины и не слишком беспокоиться вопросами по поводу ее конечной цели. Вся практическая мощь данной замечательной абстракции проявится только к концу следующей главы, а пока достаточно сказать, что любую программу, написанную на любом языке программирования высокого уровня, можно преобразовать (транслировать) в последовательность операций над стеком.

7.2.1. Push и Pop

Центральный элемент модели стековой машины — абстрактная структура данных, называемая *стеком*. Стек — это последовательное пространство для хранения данных, которое увеличивается и уменьшается по мере необходимости. Стек поддерживает различные операции, две ключевые из которых — `push` и `pop`. Операция `push` (операция «проталкивания») добавляет значение в верхнюю часть стека, подобно тому как добавляется тарелка в верхнюю часть стопки тарелок. Операция `pop` (операция «выталкивания» или «вынимания») удаляет верхнее значение стека, в результате чего верхним элементом стека становится значение, которое было непосредственно перед удаленным. Примеры этих операций показаны на иллюстрации 7.2. Обратите внимание, что логика операций `push/pop` соответствует модели «последним пришел — первым вышел» (*last-in-first-out*, LIFO): значение, вынимаемое из стека операцией `pop`, всегда является последним, которое было добавлено в стек операцией `push`. Как оказалось, такая логика доступа идеально подходит для целей трансляции и выполнения программ, но для полного раскрытия этой идеи потребуются две главы.

Как показано на иллюстрации 7.2, наша абстракция ВМ включает в себя стек, а также последовательностный сегмент памяти, похожий на оперативную память. Обратите внимание, что доступ к стеку отличается от доступа к обычной памяти. Во-первых, стек доступен только с его верхней части, тогда как обычная память подразумевает прямой

и индексированный доступ к любому значению в ней. Во-вторых, чтение значения из стека — это операция с потерями: прочитать возможно лишь верхнее значение, и единственный способ доступа к нему подразумевает *удаление* его из стека (хотя некоторые модели стека допускают также операцию *peek*, позволяющую прочитать значение без его удаления). В отличие от этого, акт чтения значения из обычной памяти не оказывает никакого влияния на состояние памяти. Наконец, запись в стек подразумевает добавление верхнего значения без изменения других значений в стеке. В отличие от этого, запись элемента в ячейку обычной памяти — это операция с потерями, поскольку она уничтожает предыдущее значение ячейки.

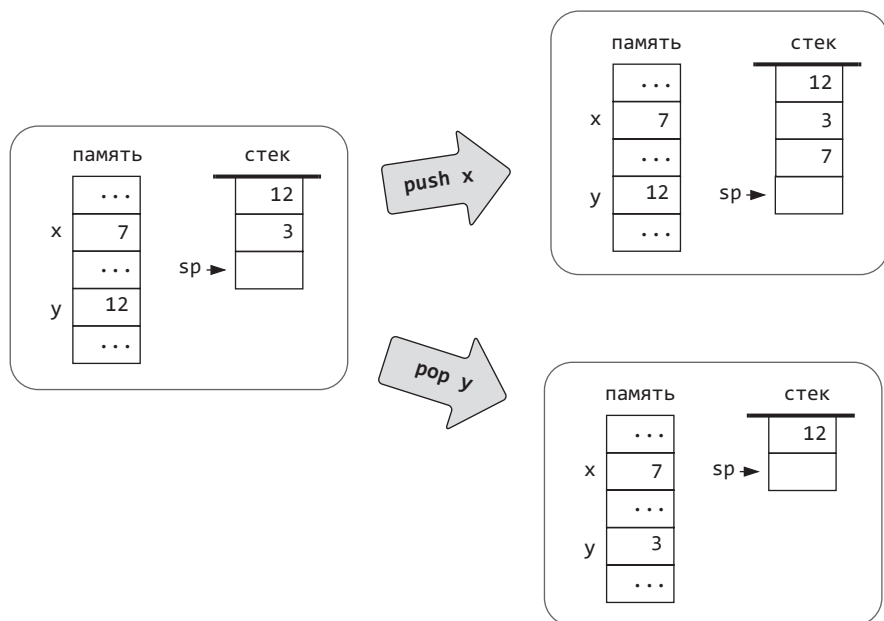
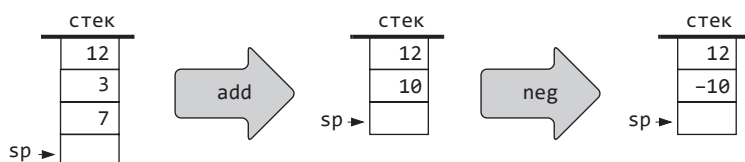


Иллюстрация 7.2. Пример обработки стека с демонстрацией двух элементарных операций `push` и `pop`. Конструкция состоит из двух структур данных: сегмента памяти типа RAM и стека. По принятой условности стек изображается «вверх ногами» («верхнее» значение стека находится внизу). На место, следующее за верхним значением стека, ссылается указатель стека (*stock pointer*), или *sp*. Символами *x* и *y* обозначены два произвольных участка памяти.

7.2.2. Стековая арифметика

Рассмотрим операцию общего вида $x \text{ op } y$, где оператор op применяется к операндам x и y , например, $7 + 5$, $3 - 8$ и т. д. В стековой машине каждая операция $x \text{ op } y$ выполняется следующим образом: сначала с вершины стека выталкиваются операнды x и y ; затем вычисляется значение $x \text{ op } y$; наконец, вычисленное значение помещается на вершину стека. Аналогичным образом унарная операция $\text{op } x$ выполняется посредством выталкивания x с вершины стека, вычисления значения $\text{op } x$ и помещения этого значения на вершину стека. Например, вот как обрабатываются сложение (*add*) и отрицание (*neg*).



Стековая обработка общих арифметических выражений продолжает ту же идею. Рассмотрим для примера выражение $d = (2 - x) + (y + 9)$, взятое из некоей высокоуровневой программы. Обработка этого выражения с помощью стека показана на иллюстрации 7.3а. На иллюстрации 7.3б показана обработка логического выражения согласно той же схеме.

Обратите внимание, что относительно стека каждая арифметическая или логическая операция влечет за собой замену операндов операции на ее результат, но не затрагивает остальную часть стека. Это похоже на то, как люди выполняют вычисления в уме с помощью кратковременной памяти. Например, как вычислить значение $3 \times (11 + 7) - 6$? Начнем с того, что мысленно «вытолкнем» из выражения 11 и 7 и вычислим $11 + 7$. Затем мы подставляем полученное значение обратно в выражение: $3 \times 18 - 6$. В результате $(11 + 7)$ заменилось на 18, а остальная часть выражения осталась прежней. Теперь мы можем продолжать выполнять аналогичные умственные операции по «выталкиванию» и «подстановке», пока не сведем выражение к одному значению.

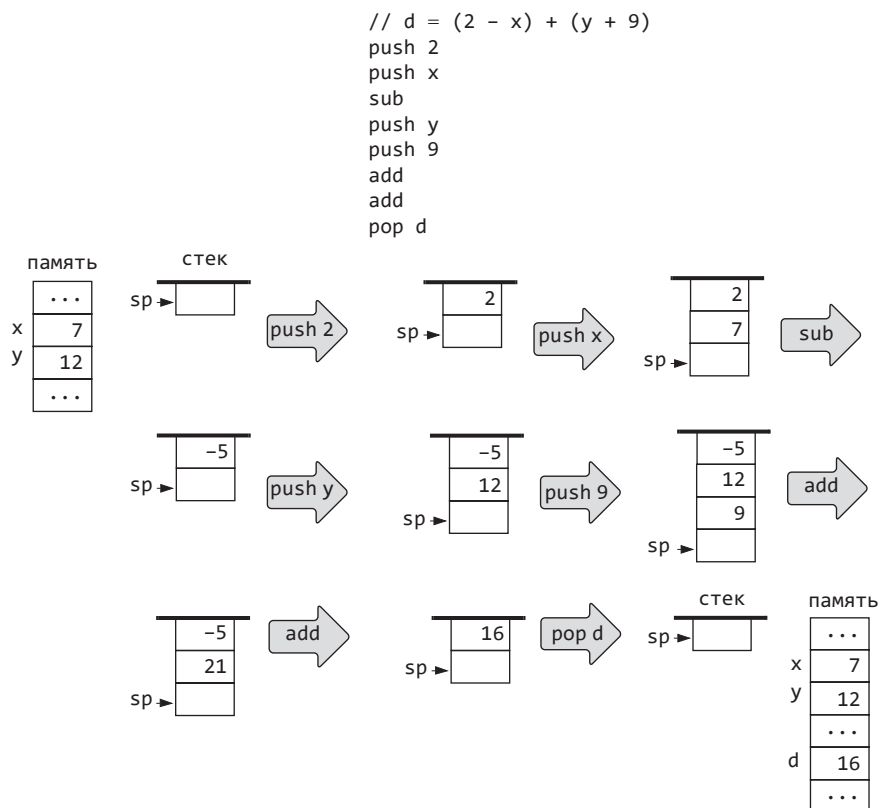


Иллюстрация 7.3а. Стековая обработка арифметического выражения.

Эти примеры иллюстрируют важное преимущество стековых машин: любое арифметическое и логическое выражение — независимо от его сложности — можно систематически преобразовать в последовательность простых операций над стеком и выполнить с помощью этой последовательности. Поэтому можно написать компилятор, переводящий высокоуровневые арифметические и логические выражения в последовательность стековых команд, что мы и сделаем в главах 10–11. После того как высокоуровневые выражения будут сведены к набору стековых команд, их можно будет выполнить с помощью реализации стековой машины.

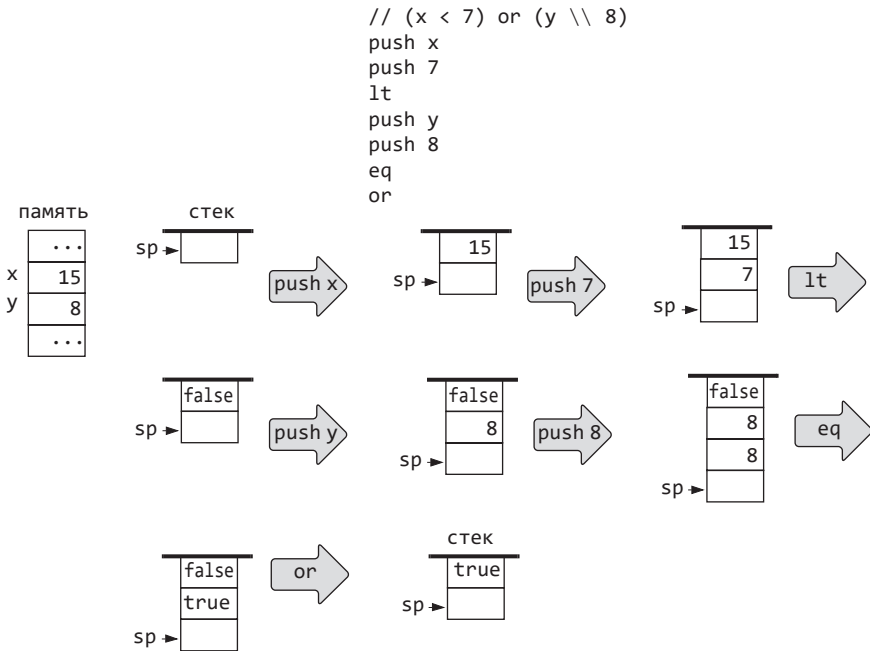


Иллюстрация 7.36. Стековая обработка логического выражения.

7.2.3. Сегменты виртуальной памяти

До сих пор в наших примерах обработки стековых команд команды push/pop иллюстрировались концептуально, с использованием синтаксиса push *x* и pop *y*, где *x* и *y* абстрактно представляли собой произвольные участки памяти. Перейдем теперь к формальному описанию команд push и pop.

В языках высокого уровня используются символьные переменные, такие как *x*, *y*, *sum*, *count* и т. д. Если язык объектный, то каждая такая переменная может быть *статической* переменной уровня класса, *полем* объекта на уровне экземпляра или *локальной* или *аргументной* переменной на уровне метода. В языках виртуальных машин, таких как Java JVM и в нашей собственной модели VM, символьных переменных не бывает. Вместо этого переменные представлены в виде записей в сегментах виртуальной памяти, имеющих имена вроде *static*, *this*, *local* и *argument*.

В частности, как мы увидим в последующих главах, компилятор отображает первую, вторую, третью и последующие статические переменные, встречаемые в программе языка высокого уровня, в виде `static 0`, `static 1`, `static 2` и т. д. Другие типы переменных аналогичным образом сопоставляются с сегментами `this`, `local` и `argument`. Например, если локальная переменная `x` и поле `y` были сопоставлены с сегментами `local 1` и `this 3` соответственно, то оператор высокого уровня типа `let x = y` будет переведен компилятором как `push this 3` с последующим `pop local 1`. В целом наша модель ВМ содержит восемь сегментов памяти, имена и роли которых перечислены на иллюстрации 7.4.

Заметим мимоходом, что разработчикам реализаций ВМ не нужно заботиться о том, как компилятор сопоставляет символьные переменные с сегментами виртуальной памяти. Мы подробно рассмотрим эти вопросы при разработке компилятора в главах 10–11. Пока же заметим, что команды ВМ обращаются ко всем сегментам виртуальной памяти совершенно одинаково: с помощью имени сегмента, за которым следует неотрицательный индекс.

<i>Сегмент</i>	<i>Роль</i>
<code>argument</code>	Аргумент функции
<code>local</code>	Местные переменные функции
<code>static</code>	Статические переменные, видимые функцией
<code>constant</code>	Константные значения 0, 1, 2, 3, ..., 32767
<code>this</code>	Описана в последующих главах
<code>that</code>	Описана в последующих главах
<code>pointer</code>	Описана в последующих главах
<code>temp</code>	Описана в последующих главах

Иллюстрация 7.4. Сегменты виртуальной памяти.

7.3. Спецификация ВМ, часть I

Наша модель ВМ основана на стеке: все операции ВМ берут свои операнды из стека и хранят свои результаты в стеке. Существует только один тип данных: знаковое 16-битное целое число. *Программа ВМ* — это последовательность команд ВМ, которые делятся на четыре категории.

- Команды *push/pop*, передающие данные между стеком и сегментами памяти.
- Арифметико-логические команды, осуществляющие арифметические и логические операции.
- Команды ветвления, позволяющие производить операции условного и безусловного ветвления.
- Команды функций, позволяющие осуществлять вызов и возврат функций.

Спецификация и реализация этих команд занимает две главы. В данной главе мы сосредоточимся на арифметико-логических командах и командах *push/pop*, а в следующей завершим спецификацию остальных команд.

Комментарии и пробелы: строки, начинающиеся с *//*, считаются комментариями и игнорируются. Пустые строки разрешены и игнорируются.

Команды *push/pop*

- | | |
|----------------------------|---|
| <i>push сегмент индекс</i> | Перемещает значение <i>сегмент[индекс]</i> в стек, где <i>сегмент</i> — <i>argument, local, static, constant, this, that, pointer</i> или <i>temp</i> , а <i>индекс</i> — неотрицательное целое число. |
| <i>pop сегмент индекс</i> | Удаляет верхнее значение стека и сохраняет его в <i>сегмент[индекс]</i> , где <i>сегмент</i> — <i>argument, local, static, constant, this, that, pointer</i> или <i>temp</i> , а <i>индекс</i> — неотрицательное целое число. |

Арифметико-логические команды

- Арифметические команды: add, sub, neg .
- Команды сравнения: eq, gt, lt .
- Логические команды: and, or, not .

Команды add, sub, eq, gt, lt, and и or имеют два неявных операнда. Для выполнения каждой команды реализация ВМ забирает с вершины стека два значения, вычисляет указанную операцию с ними и помещает полученное значение обратно в стек. Под *неявными операндами* подразумевается, что операнды не являются частью синтаксиса команды: поскольку команда всегда оперирует двумя верхними значениями стека, нет необходимости их указывать. Остальные команды neg и not имеют один неявный операнд и работают аналогичным образом. Подробности указаны на иллюстрации 7.5.

Команда	Вычисляет	Комментарий
add	$x + y$	целочисленное сложение (с дополнительным кодом)
sub	$x - y$	целочисленное вычитание (с дополнительным кодом)
neg	$-y$	арифметическое отрицание (с дополнительным кодом)
eq	$x == y$	равность (эквивалентность)
gt	$x > y$	больше, чем
lt	$x < y$	меньше, чем
and	$x \text{ And } y$	побитовое And
or	$x \text{ Or } y$	побитовое Or
not	$\text{Not } y$	побитовое Not

стек

...

x

y

sp →

Иллюстрация 7.5. Арифметико-логические команды языка ВМ.

7.4. Реализация

До этого мы описывали виртуальную машину как абстракцию. Если мы хотим использовать эту виртуальную машину по-настоящему, ее нужно реализовать на какой-то реальной платформе. Существует

несколько вариантов реализации, из которых опишем один: *транслятор ВМ*. Транслятор ВМ — это программа, которая переводит команды ВМ в инструкции машинного языка. Написание такой программы связано с двумя основными задачами. Во-первых, необходимо решить, как представлять стек и сегменты виртуальной памяти на целевой платформе. Во-вторых, мы должны перевести каждую команду ВМ в последовательность инструкций низкого уровня, которые могут выполняться на целевой платформе.

Предположим, например, что целевая платформа — это типичная машина фон Неймана. В этом случае стек виртуальной машины можно реализовать с помощью выделенного блока памяти в оперативной (RAM) памяти хоста. Нижний участок этого блока памяти будет иметь фиксированный базовый адрес, а его верхний участок будет меняться по мере роста и уменьшения стека. Таким образом, при наличии фиксированного базового адреса `stockBase` мы можем управлять стеком, отслеживая только одну переменную: *указатель стека*, или `SP`, содержащую адрес записи в оперативной памяти, следующей сразу за самым верхним значением стека. Для инициализации стека устанавливаем значение `SP` равным `stockBase`. С этого момента каждая команда `push x` может быть реализована псевдокодowymi операциями $RAM[SP] = x$ с последующей $SP++$, а каждая команда `pop x` может быть реализована операциями $SP--$ с последующей $x = RAM[SP]$.

Предположим, что хост-платформой является компьютер Hack и что мы решили закрепить базу стека по адресу 256 в оперативной памяти Hack. В данном случае транслятор ВМ может начать с генерации ассемблерного кода, реализующего операцию $SP = 256$, то есть `@256, D = A, @SP, M = D`. С этого момента транслятор ВМ может обрабатывать каждую команду `push x` и `pop x`, генерируя ассемблерный код, реализующий операции $RAM[SP++] = x$ и $x = RAM[--SP]$ соответственно.

Ориентируясь на это, теперь рассмотрим реализацию арифметико-логических команд виртуальной машины `add`, `sub`, `neg` и т. д. Удобно, что все эти команды имеют совершенно одинаковую логику доступа: выталкивание операндов команды из стека, выполнение простого вычисления и помещение результата в стек. Это означает, что нужно только понять, как реализовать команды *push* и *pop* виртуальной

машины, после чего реализация арифметико-логических команд виртуальной машины не составит труда.

7.4.1. Стандартное отображение ВМ на платформе Nask, часть I

До сих пор в этой главе все рассуждения носили абстрактный характер и не делалось никаких предположений относительно целевой платформы, на которой будет реализована наша виртуальная машина. Что касается виртуальных машин, то в этой платформенной независимости и заключается весь смысл: нет смысла привязывать абстрактную машину к какой-то конкретной аппаратной платформе, именно потому что она должна потенциально работать на любой платформе, включая те, которые еще не построены или не изобретены.

Конечно, в какой-то момент приходится реализовывать абстракцию ВМ на конкретной аппаратной платформе (например, на одной из целевых платформ, упомянутых на рисунке 7.1). Как же это сделать в нашем случае? В принципе, мы можем делать все что угодно, лишь бы в итоге реализовать абстракцию ВМ точно и эффективно. Тем не менее архитекторы ВМ обычно публикуют основные рекомендации, так называемые *стандартные отображения (привязки)*, для реализации на различных аппаратных платформах. Ориентируясь на эту практику, в оставшейся части данного раздела мы предоставим стандартное отображение нашей абстрактной ВМ на компьютере Nask. В дальнейшем будем использовать термины *реализация ВМ* и *транслятор ВМ* как взаимозаменяемые.

Программа ВМ: полное определение *программы ВМ* будет представлено в следующей главе. Пока же мы рассматриваем программу ВМ как последовательность команд ВМ, хранящихся в текстовом файле с именем *FileName.vm* (первый символ имени файла должен быть заглавной буквой, а расширение — *vm*). Транслятор ВМ должен прочитать каждую строку в файле, обработать ее как команду ВМ и перевести как одну или несколько команд на языке Nask. Полученный результат — последовательность инструкций ассемблера Nask — сохраняется в текстовом файле с именем *FileName.asm* (имя файла идентично

имени исходного файла; расширение должно быть `asm`). При трансляции ассемблером Nask в двоичный код или при запуске как есть на эмуляторе процессора Nask этот `.asm` файл должен выполнять семантику, предусмотренную исходной программой VM.

Тип данных: абстракция VM имеет только один тип данных: знаковое целое число. Этот тип реализован на платформе Nask как 16-битное значение с дополнительным кодом (согласно методу «дополнения до двух»). Булевы значения VM *true* и *false* представлены как `-1` и `0` соответственно.

Использование ОЗУ: ОЗУ (RAM) компьютера-хоста Nask состоит из 32К 16-битных слов. Реализация виртуальной машины должна использовать верхнюю часть этого адресного пространства следующим образом.

<i>Адрес RAM</i>	<i>Использование</i>
0–15	Шестнадцать виртуальных регистров, описанных ниже
16–255	Статические переменные
256–2047	Стек

Напомним, что, согласно спецификации машинного языка Nask (глава 6), адреса оперативной памяти с 0 по 4 можно обозначать символами `SP`, `LCL`, `ARG`, `THIS` и `THAT`. Эта условность была добавлена в язык ассемблера специально, чтобы в дальнейшем помочь разработчикам реализации VM писать читабельный код. Предполагаемое использование этих адресов в реализации VM следующее.

<i>Имя</i>	<i>Расположение</i>	<i>Использование</i>
<code>SP</code>	<code>RAM[0]</code>	Указатель стека (<i>sp</i>): адрес памяти, следующий за адресом памяти, содержащим самое верхнее значение стека
<code>LCL</code>	<code>RAM[1]</code>	Базовый адрес сегмента <code>local</code>
<code>ARG</code>	<code>RAM[2]</code>	Базовый адрес сегмента <code>argument</code>

<i>Имя</i>	<i>Расположение</i>	<i>Использование</i>
THIS	RAM[3]	Базовый адрес сегмента <code>this</code>
THAT	RAM[4]	Базовый адрес сегмента <code>that</code>
TEMP	RAM[5–12]	Содержит сегмент <code>temp</code>
R13	RAM[13–15]	Если ассемблерному коду, генерируемому транслятором ВМ, понадобятся переменные, он может воспользоваться этими регистрами
R14		
R15		

Говоря *базовый адрес* сегмента, мы имеем в виду физический адрес в оперативной памяти машины-хоста. Например, если хотим отобразить сегмент `local` в физическом сегменте оперативной памяти, начинающемся по адресу 1017, то можем написать код `NACK`, устанавливающий для `LCL` значение 1017. Вскользь отметим, что выбор места расположения сегментов виртуальной памяти в оперативной памяти хоста — тонкий вопрос. Например, каждый раз при выполнении функции приходится выделять место в оперативной памяти для размещения ее сегментов памяти `local` и `argument`. А когда функция вызывает другую функцию, нужно сохранить значения в этих сегментах и выделить дополнительное пространство ОЗУ для представления сегментов вызванной функции и т. д. Как можно гарантировать, что эти выделенные сегменты не переполнятся и не затронут другие зарезервированные области ОЗУ? Мы рассмотрим эти проблемы управления памятью в следующей главе, когда будем реализовывать команды вызова и возврата функций в языке ВМ.

Но пока что ни один из этих вопросов распределения памяти не должен нас беспокоить. Условимся считать, что `SP`, `ARG`, `LCL`, `THIS` и `THAT` уже инициализированы по некоторым разумным адресам в оперативной памяти хоста (то есть в эти регистры записаны значения, соответствующие адресам памяти). Обратите внимание, что реализации ВМ никогда не видят этих адресов. Вместо того чтобы обращаться к ним напрямую, они манипулируют ими символически, используя имена указателей. Предположим, что мы хотим поместить в стек значение регистра `D`. Эту операцию можно реализовать с помощью логики $RAM[SP++] = D$, которую в ассемблере `NACK` можно выразить как $@SP, A = M, M = D, @SP, M = M + 1$. Этот код прекрасно

выполнит операцию *push*, не зная при этом ни местоположения стека в оперативной памяти хоста, ни текущего значения указателя стека.

Мы рекомендуем потратить несколько минут на то, чтобы переварить только что показанный ассемблерный код. Если вы не поняли его, вам необходимо освежить свои знания о манипулировании указателями в языке ассемблера Hack (раздел 4.3, пример 3). Эти знания являются необходимым условием для разработки транслятора VM, поскольку перевод каждой команды VM подразумевает генерацию кода на языке ассемблера Hack.

Привязка сегментов памяти

Local, argument, this, that: в следующей главе мы обсудим, как реализация VM динамически привязывает эти сегменты к оперативной памяти хоста. Пока же достаточно знать, что базовые адреса этих сегментов хранятся в регистрах LCL, ARG, THIS и THAT соответственно. Поэтому любой доступ к *i*-й записи виртуального сегмента (в контексте команды VM «*push/pop сегментИмя i*») нужно перевести в ассемблерный код, который обращается к адресу (*base + i*) в оперативной памяти, где *base* («базовый адрес») — один из указателей LCL, ARG, THIS или THAT.

Указатель (pointer): в отличие от виртуальных сегментов, описанных выше, сегмент указателя *pointer* содержит ровно два значения и привязан непосредственно к участкам 3 и 4 оперативной памяти. Напомним, что эти участки в оперативной памяти также называются соответственно THIS и THAT. В результате семантика сегмента указателя выглядит следующим образом. Любое обращение к *pointer 0* должно привести к обращению к указателю THIS, а любое обращение к *pointer 1* должно привести к обращению к указателю THAT. Например, команда *pop pointer 0* присваивает имя THIS выталкиваемому значению, а команда *push pointer 1* должна поместить в стек текущее значение THAT. Эта своеобразная семантика станет более понятной, когда мы будем писать компилятор в главах 10–11, так что дождитесь момента.

Temp: этот сегмент из 8 слов также фиксирован и привязан непосредственно к участкам 5–12 оперативной памяти. С учетом этого любой доступ к `temp i`, где i варьируется от 0 до 7, должен быть переведен в ассемблерный код, который обращается к участку $5 + i$ в оперативной памяти.

Constant: этот сегмент виртуальной памяти поистине виртуален, поскольку не занимает никакого физического пространства оперативной памяти. Вместо этого реализация ВМ обрабатывает любой доступ к константе i , просто предоставляя константу i . Например, команда `push constant 17` переводится в ассемблерный код, который помещает в стек значение 17.

Static: статические переменные привязываются к адресам с 16 по 255 оперативной памяти хоста. Транслятор ВМ может автоматически реализовывать такую привязку следующим образом. Каждая ссылка на `static i` в программе ВМ, хранящейся в файле `Foo.vm`, может быть переведена в ассемблерный символ `Foo.i`. Согласно спецификации машинного языка *Hack* (глава 6), ассемблер *Hack* размещает эти символические переменные в оперативной памяти хоста, начиная с адреса 16. В результате статические переменные, появляющиеся в программе ВМ, привязываются к адресам 16 и далее в том порядке, в котором они появляются в коде ВМ. Предположим, что программа ВМ начинается с команд `push constant 100, push constant 200, pop static 5, pop static 2`. Описанная выше схема трансляции приведет к тому, что `static 5` и `static 2` будут привязаны к адресам ОЗУ 16 и 17 соответственно.

Такая реализация статических переменных несколько коварна, но работает хорошо. Благодаря ей статические переменные разных файлов ВМ могут сосуществовать, не смешиваясь, поскольку генерируемые ими символы *FileName.i* имеют уникальные префиксные имена с указанием файлов. В заключение отметим, что, поскольку стек начинается с адреса 256, данная реализация ограничивает количество статических переменных в программе *Jack* до $255 - 16 + 1 = 240$.

Символы языка ассемблера: обобщим все специально упомянутые выше символы. Предположим, что программа VM, которую мы должны перевести, содержится в файле с именем `Foo.vm`. Трансляторы VM, соответствующие *стандартному отображению VM на платформе Hack*, генерируют ассемблерный код, в котором используются следующие символы: `SP`, `LCL`, `ARG`, `THIS`, `THAT` и `Foo.i`, где *i* — неотрицательное целое число. Если необходимо сгенерировать код, использующий переменные для временного хранения, трансляторы VM могут воспользоваться символами `R13`, `R14` и `R15`.

7.4.2. Эмулятор виртуальной машины

Один из относительно простых способов реализации виртуальной машины — создание высокоуровневой программы, представляющей стек и сегменты памяти и реализующей все команды VM с помощью высокоуровневого программирования. Например, если представить стек с помощью достаточно большого массива с именем `stock`, то операции *push* и *pop* можно реализовать непосредственно с помощью высокоуровневых операторов типа `stock[SP++] = x` и `x = stock[--SP]` соответственно. Сегменты виртуальной памяти также можно обрабатывать с помощью массивов.

Если же нам захочется еще больших «наворотов», то можно дополнить программу графическим интерфейсом, позволяющим пользователям экспериментировать с командами виртуальной машины и визуально наблюдать их влияние на образы стека и сегментов памяти. В комплект программ «От Nand до “Тетриса”» входит как раз один из таких эмуляторов, написанный на языке Java (см. иллюстрация 7.6). Эта удобная программа позволяет загружать и выполнять код VM в исходном виде и визуально наблюдать во время имитации выполнения за тем, как команды VM влияют на состояние эмулируемых стека и сегментов памяти. Кроме того, эмулятор показывает привязку стека и сегментов памяти к оперативной памяти хоста, а также показывает, как меняется состояние оперативной памяти при выполнении команд VM. В общем, поставляемый эмулятор VM — это очень классная программа: попробуйте ее!

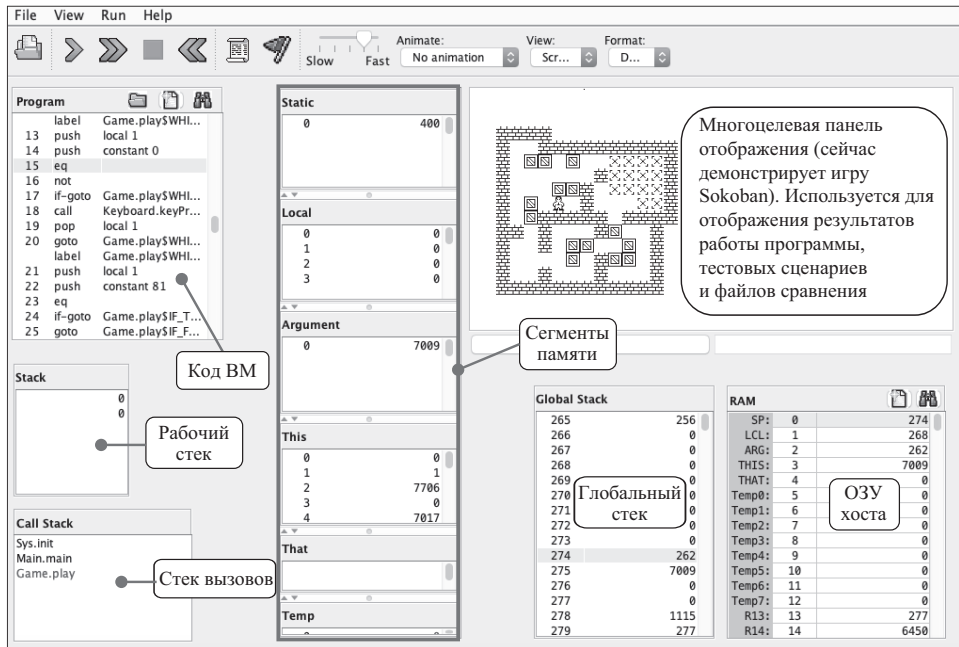


Иллюстрация 7.6. Эмулятор виртуальной машины, поставляемый вместе с программным комплектом «От Nand до “Тетриса”».

7.4.3. Предложения по реализации ВМ

Использование: транслятор ВМ принимает один аргумент командной строки, как показано ниже:

```
prompt> VMTTranslator source,
```

где *source* — имя файла вида *ProgName.vm*. Имя файла может содержать путь к файлу. Если путь не указан, транслятор ВМ работает с текущей папкой. Первый символ в имени файла должен быть заглавной буквой, а расширение *vm* является обязательным. Файл содержит последовательность одной или нескольких команд ВМ. В результате работы транслятор создает выходной файл с именем *ProgName.asm*, содержащий ассемблерные команды, реализующие команды ВМ. Выходной файл *ProgName.asm* хранится в той же папке, что и входной. Если файл *ProgName.asm* уже существует, он будет перезаписан.

Структура программы

Мы предлагаем реализовать транслятор ВМ с помощью трех модулей: основной программы VMTranslator, модуля Parser (парсера) и модуля CodeWriter (составителя кода). Задача Parser — извлечь смысл из каждой команды ВМ, то есть понять, что команда хочет сделать. Работа CodeWriter заключается в том, чтобы перевести понятую команду ВМ в ассемблерные команды, реализующие требуемые операции на платформе Hack. VMTranslator управляет процессом перевода.

Parser

Этот модуль осуществляет синтаксический разбор одного файла .vm. Парсер прочитывает команды ВМ, раскладывает их на различные компоненты и предоставляет удобный доступ к ним. Кроме того, он игнорирует все пробелы и комментарии. Парсер предназначен для обработки всех команд ВМ, включая команды *ветвления* и *функций*, что будет реализовано в главе 8.

Например, если текущая команда — `push local 2`, то вызов `arg1()` и `arg2()` вернет, соответственно, "local" и 2. Если текущей командой является `add`, то вызов `arg1()` вернет "add", а `arg2()` не будет вызываться.

<i>Процедура</i>	<i>Аргументы</i>	<i>Возвращает</i>	<i>Функция</i>
Конструктор/ инициализатор	Файл ввода / поток	—	Открывает входной файл / поток и подготавливает его к разбору
hasMoreLines	—	Булево значение (boolean)	Есть ли еще строки на входе?
advance	—	—	Читает со входа следующую команду и делает ее текущей. Эту рутину следует вызывать, только если hasMoreLines равно true. Изначально текущей команды нет
commandType	—	C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL (константа)	Возвращает константу, представляющую тип текущей команды. Если текущая команда является арифметико- логической, возвращает C_ARITHMETIC
arg1	—	Строку	Возвращает первый аргумент текущей команды. В случае C_ARITHMETIC возвращается сама команда (add, sub и т. д.). Не должна вызываться, если текущей командой является C_RETURN
arg2	—	Целое число (int)	Возвращает второй аргумент текущей команды. Следует вызвать только в том случае, если текущей командой является C_PUSH, C_POP, C_FUNCTION или C_CALL

CodeWriter

Этот модуль преобразует разложенные команды VM в ассемблерный код Hack.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Файл ввода / поток	—	Открывает входной файл / поток и подготавливает его к разбору
writeArithmetic	Команда (строка)	—	Записывает в выходной файл ассемблерный код, реализующий заданную арифметико-логическую команду
writePushPop	Команда (C_PUSH или C_POP), сегмент (строка), индекс (int)	—	Записывает в выходной файл ассемблерный код, реализующий заданную команду push или pop
Close		—	Закрывает входной файл / поток

Например, вызов `writePushPop(C_PUSH, "local", 2)` приведет к генерации ассемблерных команд, реализующих команду VM `push local 2`. Другой пример: вызов `WriteArithmetic("add")` приведет к генерации ассемблерных команд, которые извлекают два самых верхних элемента из стека, складывают их и помещают результат в стек.

VMTranslator

Это основная программа, управляющая процессом перевода и обращающаяся к службам Parser и CodeWriter. Программа получает имя входного исходного файла — допустим, *Prog.vm* — из аргумента командной строки. Далее она конструирует парсер для разбора входного

файла *Prog.vm* и создает выходной файл *Prog.asm*, в который будет записывать переведенные ассемблерные команды. Затем программа переходит в цикл, который итеративно просматривает во входном файле команды VM. Для каждой команды программа использует службы *Parser* и *CodeWriter* для разбивки команды на поля и для последующей генерации на их основе ассемблерных команд. Эти ассемблерные команды записываются в выходной файл *Prog.asm*.

Мы не предоставляем API для этого модуля, предлагая вам реализовать его по своему усмотрению.

Советы по реализации

1. Приступая к переводу команды VM, например `push local 2`, предусмотрите генерацию и комментарии к ассемблерному коду вида `// push local 2`. Такие комментарии помогут вам прочесть сгенерированный код и при необходимости отладить ваш транслятор.
2. Почти каждая команда VM должна помещать данные в стек или вынимать их из стека. Поэтому ваши рутины `write Xxx` неизбежно будут выводить похожие ассемблерные команды одну за другой. Во избежание написания повторяющегося кода подумайте об использовании частных процедур (иногда называемых *вспомогательными методами*), которые будут генерировать эти часто используемые фрагменты кода.
3. Как объяснялось в главе 6, каждую программу на машинном языке рекомендуется завершать бесконечным циклом. Поэтому подумайте о написании частной рутины, записывающей код бесконечного цикла на языке ассемблера. Вызовите эту процедуру один раз, когда закончите трансляцию всех команд VM.

7.5. Проект

По сути, вам нужно написать программу, которая прочитывает по одной за раз команды VM и переводит эти команды на язык ассемблера Hack. Как, например, нужно обработать команду VM `push local 2`?

Совет: нужно написать несколько ассемблерных команд, которые, помимо прочего, манипулируют указателями SP и LCL. Придумать последовательность команд на языке ассемблера Hack, реализующую каждую из арифметико-логических команд VM и команд *push/pop*, — вот в чем суть этого проекта. Именно в этом и заключается генерация кода.

Рекомендуем вам начать с написания и тестирования фрагментов ассемблерного кода на бумаге. Нарисуйте сегмент оперативной памяти, начертите таблицу, по которой будете отслеживать, например, значения SP и LCL, и инициализируйте эти переменные по произвольным адресам памяти. Отследите на бумаге набор ассемблерных команд, которые, по вашему мнению, реализуют команду *push local 2*. Правильно ли они влияют на стек и на сегменты *local* (с точки зрения ОЗУ)? Не забыли ли вы обновить указатель стека? Продолжайте в том же духе. Когда у вас появится уверенность в том, что ваши фрагменты ассемблерного кода правильно выполняют свою работу, можете почти полностью переложить задачу по генерации на модуль *CodeWriter*.

Так как ваш транслятор VM должен выдавать код на языке ассемблера, будет неплохо поупражняться в его анализе и написании. Лучше всего для этого будет просмотреть примеры программ на ассемблере из главы 4 и программы, которые вы написали в проекте 4. При необходимости обращайтесь к документации по языку ассемблера в разделе 4.2.

Задача: создать базовый транслятор VM, предназначенный для реализации арифметико-логических и *push/pop*-команд языка VM. Для этой версии транслятора VM предполагается, что исходный код VM не содержит ошибок. Проверку и обработку ошибок можно будет добавить в последующие версии транслятора VM, но в проект 7 они не входят.

Ресурсы: вам понадобятся два инструмента: язык программирования, на котором вы будете реализовывать транслятор VM, и *эмулятор ЦПУ*, находящийся в папке *nand2tetris/tools*. Эмулятор ЦПУ позволит вам выполнить и протестировать ассемблерный код, сгенерированный вашим транслятором. Если сгенерированный код в эмуляторе

ЦПУ работает правильно, будем считать, что ваш транслятор ВМ работает так, как ожидалось. Это лишь частичная проверка транслятора, но для наших целей этого будет достаточно.

Еще один инструмент, который пригодится в данном проекте, — эмулятор ВМ, также находящийся в папке `nand2tetris/tools`. Мы рекомендуем использовать эту программу для выполнения прилагаемых тестовых программ и наблюдения за тем, как код ВМ влияет на (смоделированные) состояния стека и сегментов виртуальной памяти. Предположим, например, что тестовая программа помещает несколько констант в стек, а затем удаляет их из стека и переносит в сегмент `local`. Можно запустить тестовую программу на эмуляторе виртуальной машины, проследить, как стек увеличивается и уменьшается, и посмотреть, как сегмент `local` заполняется значениями. Это поможет вам понять, какие действия должен генерировать транслятор ВМ, прежде чем приступить к его реализации.

Контракт: написать транслятор ВМ с языка ВМ на ассемблерный язык Наск, соответствующий спецификации ВМ, приведенной в разделе 7.3, и стандартному отображению ВМ на платформе Наск, описанному в разделе 7.4.1. Воспользуйтесь своим транслятором для перевода предоставленных тестовых программ ВМ и получите соответствующие им программы, написанные на языке ассемблера Наск. При выполнении на прилагаемом эмуляторе ЦПУ сгенерированные вашим транслятором программы на языке ассемблера должны обеспечивать результаты, предусмотренные прилагаемыми тестовыми сценариями и файлами сравнения.

Этапы тестирования и реализации

Мы предоставляем пять тестовых программ ВМ. По мере работы над транслятором рекомендуем тестировать их в том порядке, в котором они приведены. Так вы будете двигаться в верном направлении и постепенно увеличивать возможности своего транслятора по генерации кода в соответствии с требованиями, предъявляемыми каждой тестовой программой.

SimpleAdd: эта программа помещает две константы в стек и складывает их. Проверяет, как ваша реализация обрабатывает команды `push constant i` и `add`.

StockTest: помещает в стек некоторые константы и проверяет, как ваша реализация обрабатывает все арифметико-логические команды.

BasicTest: выполняет команды `push`, `pop` и арифметические команды с использованием сегментов памяти `constant`, `local`, `argument`, `this`, `that` и `temp`. Проверяет, как ваша реализация обрабатывает эти сегменты памяти (до этого вы уже имели дело с `constant`).

PointerTest: выполняет команды `push`, `pop` и арифметические команды, используя сегменты памяти `pointer`, `this` и `that`. Проверяет, как ваша реализация обрабатывает сегмент `pointer`.

StaticTest: выполняет команды `push`, `pop` и арифметические команды, используя константы и сегмент памяти `static`. Проверяет, как ваша реализация работает с сегментом `static`.

Инициализация: для того чтобы любая транслированная программа ВМ начала выполняться, она должна содержать код запуска, который заставляет сгенерированный ассемблерный код начать выполнение на хост-платформе. А прежде чем этот код начнет выполняться, реализация виртуальной машины должна закрепить базовые адреса стека и сегментов виртуальной памяти в выбранных местах оперативной памяти. Оба вопроса — инициализация кода запуска и инициализация сегментов — описываются и предлагаются для реализации в следующей главе. Сложность здесь заключается в том, что эти инициализации нам нужны для выполнения тестовых программ в данном проекте. Но вам не стоит пока беспокоиться об этих деталях, потому что все инициализации, необходимые для нашего проекта, выполняются «вручную» прилагаемыми тестовыми скриптами.

Тестирование/отладка: мы предлагаем в комплекте пять наборов тестовых программ, тестовых скриптов и файлов сравнения. Для каждой тестовой программы `Xxx.vm` мы рекомендуем выполнять следующие шаги.

0. Воспользуйтесь сценарием `Xxx\VM .tst` для выполнения тестовой программы `Xxx.vm` на прилагаемом эмуляторе виртуальной машины. Так вы ознакомитесь с предполагаемым поведением тестовой программы. Исследуйте смоделированный стек и виртуальные сегменты и постарайтесь как можно лучше понять, что делает тестовая программа.
1. Используйте ваш частично реализованный транслятор для трансляции файла `Xxx.vm` (тестовая программа). Результатом должен быть текстовый файл с именем `Xxx.asm`, содержащий ассемблерный код `Nask`, сгенерированный вашим транслятором.
2. Просмотрите сгенерированный вашим транслятором файл `Xxx.asm`. Если в нем присутствуют заметные ошибки, отладьте и исправьте ваш транслятор.
3. Используйте прилагаемые файлы `Xxx.tst` и `Xxx.cmp` для запуска и тестирования транслированной программы `Xxx.asm` в прилагаемом эмуляторе ЦПУ. При наличии каких-либо ошибок отладьте и исправьте транслятор.

Закончив работу над данным проектом, не забудьте сохранить копию своего транслятора `VM`. В следующей главе вы будете расширять эту программу, добавляя в нее больше возможностей для обработки команд `VM`. Если в ходе работы над проектом 8 разработанный для проекта 7 код будет нарушен, вы сможете загрузить резервную копию.

Веб-версия проекта 7 доступна на сайте www.nand2tetris.org.

7.6. Перспектива

В этой главе мы начали процесс разработки компилятора для языка высокого уровня. Следуя современной практике разработки программного обеспечения, мы выбрали двухуровневую модель компиляции. На внешнем («фронтенд») уровне, который рассматривается в главах 10 и 11, код высокого уровня транслируется в промежуточный код, предназначенный для выполнения на виртуальной машине. На внутреннем («бэкенд») уровне, который рассматривается в этой и следующей главах, промежуточный код транслируется на машинный язык целевой аппаратной платформы (см. иллюстрацию 7.1).

На протяжении многих лет такая двухэтапная модель компиляции явно или неявно использовалась во многих проектах по созданию компиляторов. В конце 1970-х годов корпорации IBM и Apple представили два новаторских и феноменально успешных персональных компьютера: IBM PC и Apple II. Помимо прочих языков высокого уровня, популярным на этих ранних ПК был Pascal. К сожалению, для него пришлось разрабатывать разные компиляторы, потому что в машинах IBM и Apple использовались разные процессоры, разные машинные языки и разные операционные системы. Кроме того, IBM и Apple были конкурирующими компаниями и не были заинтересованы в том, чтобы помочь разработчикам перенести свои программы на другую машину. В результате разработчики программ, которые хотели, чтобы их приложения на языке Pascal работали на обеих линейках компьютеров, вынуждены были использовать разные компиляторы, каждый из которых предназначался для создания двоичного кода, специфичного для конкретной машины. Нельзя ли было найти лучший способ кроссплатформенной компиляции — так, чтобы, написав программу один раз, запускать ее везде?

Одним из решений этой проблемы стал ранний фреймворк виртуальной машины под названием *p-код* (*p-code*). Основная идея заключалась в том, чтобы компилировать программы на языке Pascal в промежуточный *p-код* (подобный нашему языку VM), а затем использовать одну реализацию для перевода абстрактного *p-кода*

в набор инструкций для процессора Intel x86, используемого в компьютерах IBM PC, и другую реализацию для перевода того же р-кода в набор инструкций для процессора Motorola 68000, используемого в компьютерах Apple. Тем временем другие компании разработали высокооптимизированные компиляторы Pascal, генерировавшие эффективный р-код. В результате одна и та же программа на языке Pascal могла работать практически на любой машине зарождающегося рынка ПК: независимо от того, какой компьютер использовали ваши клиенты, вы могли поставлять им точно такие же файлы р-кода, избавляя их от необходимости использовать несколько компиляторов. Конечно, вся схема строилась на предположении, что компьютер клиента оснащен клиентской реализацией р-кода (эквивалентной нашему транслятору VM). Для этого реализации р-кода свободно распространялись через Интернет, и клиентам предлагалось загрузить их на свои компьютеры. Исторически это был, пожалуй, первый случай, когда в полной мере реализовалось само понятие кроссплатформенного языка высокого уровня.

Вопрос кроссплатформенной совместимости приобрел особую актуальность после бурного роста Интернета и мобильных устройств в середине 1990-х годов. Для решения данной проблемы компания Sun Microsystems (впоследствии приобретенная компанией Oracle) попыталась разработать новый язык программирования, скомпилированный код которого потенциально мог бы работать в исходном виде на любом компьютере и цифровом устройстве, подключенном к Интернету. Язык Java, появившийся в результате этой инициативы, основывался на модели выполнения промежуточного кода, называемой *виртуальной машиной Java*, или JVM.

JVM — это спецификация, описывающая некий промежуточный язык под названием «байт-код», целевой язык компиляторов VM Java. Файлы, написанные на байт-коде, широко используются для распространения кода Java-программ через Интернет. Для выполнения этих переносимых программ клиентские ПК, планшеты и мобильные телефоны, на которые такие файлы загружаются, должны быть оснащены подходящими реализациями JVM, называемыми «средами выполнения Java» (*Java Runtime Environments*, JRE). Эти программы широко

доступны для многочисленных комбинаций процессоров и операционных систем. В настоящее время многие владельцы персональных компьютеров и мобильных телефонов регулярно пользуются такими инфраструктурными программами (JRE), не задумываясь или даже не подозревая об их существовании на своих устройствах.

Язык Python, созданный в конце 1980-х годов, также основан на двухуровневой модели трансляции, центральной частью которой является PVM (Python Virtual Machine), использующая свою собственную версию байт-кода.

В начале 2000-х годов компания Microsoft выпустила платформу .NET Framework. Центральный элемент .NET — виртуальная машина под названием *Common Language Runtime* (CLR). Согласно концепции Microsoft, программы на многих языках программирования, таких как C# и C++, можно компилировать в промежуточный код, работающий на CLR. В результате программы, написанные на разных языках, взаимодействуют между собой и совместно используют программные библиотеки общей среды выполнения. Конечно, одноуровневые компиляторы для C и C++ никуда не делись и по-прежнему широко используются, особенно в высокопроизводительных приложениях, требующих плотного и оптимизированного кода.

И действительно, один из вопросов, о которых в данной главе не было сказано ни слова, — это вопрос эффективности. Наш контракт предусматривает разработку транслятора ВМ без каких бы то ни было требований к эффективности генерируемого ассемблерного кода. Очевидно, что это серьезное упущение. Транслятор ВМ — это критически важная технология, лежащая в основе вашего ПК, планшета или мобильного телефона: если он будет генерировать надежный и эффективный низкоуровневый код, приложения будут работать на вашей машине быстро и с использованием минимума ресурсов. Поэтому оптимизация транслятора ВМ — это одна из первоочередных практических задач.

В целом существует множество методов оптимизации транслятора ВМ. Например, в языках высокого уровня часто встречаются присваивания типа `let x = y`; эти выражения переводятся компилятором в такие команды ВМ, как, например, `push local 3` и `pop`

static 1. «Умная» же реализация таких пар команд ВМ может генерировать ассемблерный код, полностью обходящий стек, что приводит к значительному увеличению производительности. Конечно, это один из многих примеров возможных оптимизаций ВМ. За прошедшие годы реализации ВМ в Java, Python и C# стали значительно более мощными и сложными.

В заключение упомянем о важнейшем компоненте, который обязательно нужно добавить к модели виртуальной машины, чтобы полностью раскрыть весь ее потенциал — об общей программной библиотеке. И действительно, виртуальная машина Java JVM поставляется со *стандартной библиотекой классов Java*, а Microsoft .NET Framework — с библиотекой классов *Framework Class Library*. Эти обширные программные библиотеки можно рассматривать как своего рода переносимые операционные системы, предоставляющие многочисленные службы, такие как управление памятью, наборы инструментов графического интерфейса, строковые функции, математические функции и т. д. Эти расширения будут описаны и построены в главе 12.

8. Виртуальная машина II: управление

Если вам кажется, что вы превосходно управляете машиной,
вы просто едете недостаточно быстро.

— Марио Андретти, чемпион по автогонкам (р. 1940)

В главе 7 было введено понятие *виртуальной машины* (ВМ), а в проекте 7 мы приступили к реализации нашей абстрактной виртуальной машины и языка ВМ на платформе Nask. Реализация эта подразумевает разработку программы, переводящей команды ВМ в ассемблерный код Nask. В частности, в предыдущей главе мы узнали, как использовать и реализовывать арифметико-логические команды ВМ и команды push/pop; в этой главе узнаем, как использовать и реализовывать команды ветвления и команды функций ВМ.

По ходу главы мы будем расширять базовый транслятор, разработанный в проекте 7, и в результате у нас получится полномасштабный транслятор ВМ для платформы Nask. Этот транслятор станет внутренним модулем компилятора, который мы разработаем в главах 10 и 11.

В любом конкурсе «Великие изобретения прикладной информатики» принцип стековой обработки неизменно занимал бы призовые места. В предыдущей главе было показано, как в виде элементарных операций над стеком можно представить арифметические и булевы выражения. Далее в данной главе мы покажем, как эта удивительно простая структура данных поддерживает такие удивительно сложные задачи, как вызов вложенных функций, передача параметров, рекурсия, а также различные задачи выделения и утилизации памяти,

необходимые для исполнения программы. Большинство программистов воспринимают такие задачи как нечто должное, ожидая, что компилятор и операционная система так или иначе как-то с ними справятся. Мы же теперь можем открыть «черный ящик» и посмотреть, как эти фундаментальные механизмы программирования реализуются на самом деле.

Система поддержки исполнения программ: для каждой компьютерной системы должна быть определена модель исполнения программ, отвечающая на важные вопросы, без ответа на которые программы работать не смогут: как начать выполнение программы, что должен делать компьютер по завершении программы, как передавать аргументы от одной функции к другой, как выделять ресурсы памяти для выполняющихся функций, как освобождать ресурсы памяти, когда они больше не нужны, и т. д.

В «От Nand до “Тетриса”» эти вопросы решаются спецификацией языка *VM* вместе со стандартным отображением спецификации на платформе *Hack*. Разрабатывая транслятор *VM* в соответствии с этими рекомендациями, мы получим в итоге исполняемую систему. В частности, транслятор *VM* будет не только переводить команды *VM* (`push`, `pop`, `add` и т. д.) в инструкции ассемблера, он также будет генерировать ассемблерный код ассемблера, реализующий оболочку, в которой выполняется программа. Ответы на все упомянутые выше вопросы — как запустить программу, как управлять поведением вызова и возврата функций и др. — будут получены посредством генерации ассемблерного кода поддержки, в который и заключен собственно код. Рассмотрим это на примере.

8.1. Высокоуровневая магия

Языки высокого уровня позволяют писать программы в виде выражений высокого уровня. Например, выражение $x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$ можно записать как `x = -b + sqrt(power(b, 2) - 4 * a * c)`, что по наглядности сравнимо с оригинальным. Обратите внимание

на разницу между примитивными операциями $+$ и $-$ и такими функциями, как `sqrt` и `power`. Первые встроены в базовый синтаксис языка высокого уровня. Вторые являются расширениями базового языка.

Такая неограниченная возможность расширения языка по своему усмотрению — одна из важнейших особенностей языков программирования высокого уровня. Конечно, в какой-то момент кто-то должен реализовать такие функции, как `sqrt` и `power`. Однако история реализации этих абстракций полностью отделена от истории их использования. Поэтому прикладные программисты могут надеяться на то, что каждая из этих функций будет так или иначе выполнена и что после ее выполнения управление так или иначе вернется к следующей операции в тексте программы. Команды ветвления наделяют язык дополнительной выразительной силой, позволяя писать условный код типа `if! (a == 0) {x = (-b + sqrt(power(b, 2) - 4 * a * c)) / (2 * a)} else {x = -c / b}`. И снова мы видим, что код высокого уровня позволяет выразить логику высокого уровня — в данном случае алгоритм решения квадратных уравнений — практически напрямую.

Современные языки программирования и в самом деле дружелюбны к программистам тем, что предлагают полезные и мощные абстракции. Однако немного смущает мысль о том, что, как бы ни был высок наш язык, он в конечном счете должен быть реализован на аппаратной платформе, которая может выполнять только примитивные машинные инструкции. Поэтому, помимо всего прочего, архитекторы компиляторов и виртуальных машин должны находить низкоуровневые решения для реализации ветвления и команд вызова и возврата функций.

Функции, служащие основой модульного программирования, представляют собой отдельные программные блоки, которым разрешено вызывать друг друга для получения результата. Например, функция `solve` («решить») может вызвать функцию `sqrt` («квадратный корень»), а `sqrt`, в свою очередь, может вызвать функцию `power` («степень»). Такая последовательность вызовов может быть сколь угодно глубокой, а также рекурсивной. Как правило, *вызывающая функция* передает аргументы *вызываемой функции* и приостанавливает свое

исполнение до тех пор, пока не завершит свое исполнение последняя функция. Вызываемая функция использует переданные аргументы для выполнения или вычисления чего-либо, а затем *возвращает* значение (которое может быть и пустым, `void`) вызывающей функции. После этого вызывающая функция возобновляет свое исполнение.

В целом же, когда одна функция вызывает другую, кто-то должен позаботиться о следующих сопутствующих этому процессу задачах.

- Сохранить *адрес возврата*, то есть адрес в коде вызывающей функции, к которому программа должна будет вернуться после исполнения вызываемой функции.
- Сохранить ресурсы памяти вызывающей функции.
- Выделить ресурсы памяти для вызываемой функции.
- Сделать передаваемые аргументы вызывающей функции доступными для вызываемой.
- Начать выполнение кода вызываемой функции.

Когда вызываемая функция завершает работу и возвращает значение, кто-то должен позаботиться о следующих задачах.

- Сделать *возвращаемое значение* доступным для вызывающей функции.
- Освободить ресурсы памяти, использованные вызываемой функцией.
- Восстановить ранее сохраненные ресурсы памяти вызывающей функции.
- Получить ранее сохраненный *адрес возврата*.
- Возобновить выполнение кода вызывающей функции, начиная с адреса возврата.

К счастью, программистам на языке высокого уровня не нужно думать обо всех этих тонкостях работы: с ними незаметно и эффективно справляется генерируемый компилятором ассемблерный код. А в двухуровневой модели компиляции эти «обязанности по ведению домашнего хозяйства», то есть обработке служебных задач, ложатся

на скрытую («бэкенд») часть компилятора — транслятор ВМ, который мы сейчас и разрабатываем. Итак, в данной главе мы, помимо всего прочего, раскроем тайны системы поддержки исполнения программ, позволяющей реализовать, пожалуй, самую важную абстракцию в искусстве программирования: *вызов функций и возврат их значений*. Но для начала рассмотрим более простую задачу — обработку команд ветвления.

8.2. Ветвление

По умолчанию компьютерные программы работают последовательно, выполняя одну команду за другой. Этот поток можно перенаправлять по различным причинам, например для начала новой итерации в цикле, и делают это с помощью команд ветвления. В низкоуровневом программировании ветвление осуществляется с помощью команд типа *goto* (перехода к указанному месту). Спецификация перехода может принимать несколько форм, самая примитивная из которых — указание физического адреса памяти команды, которую нужно выполнить следующей. Несколько более абстрактная спецификация заключается в указании символической метки (привязанной к физическому адресу памяти). В таком варианте в языке должна быть предусмотрена директива маркировки, то есть присвоения символьных меток выбранным местам в коде. В нашем языке ВМ это делается с помощью команды маркировки с синтаксисом *label symbol* (где *symbol* — произвольно выбранный символ).

Исходя из вышесказанного, язык ВМ поддерживает две формы ветвления. *Безусловное ветвление* осуществляется с помощью команды *goto symbol* со следующим значением: перейти к выполнению команды, следующей в коде сразу после команды *label symbol*. *Условное ветвление* осуществляется с помощью команды *if-goto symbol* со следующей семантикой: взять самое верхнее значение из стека; если оно не ложно, перейти к выполнению команды, следующей сразу после команды *label symbol*; в противном случае выполнить следующую команду в коде. Такое соглашение подразумевает, что перед тем,

как задавать условную команду `goto`, автор кода ВМ (например, компилятор) должен сначала задать условие. В нашем языке ВМ это делается посредством помещения в стек булева выражения. Например, компилятор, который мы будем разрабатывать в главах 10–11, преобразует `if (n < 100) goto LOOP` в последовательность команд `push n, push 100, lt, if-goto LOOP`.

Пример: рассмотрим функцию, которая получает два аргумента x и y и возвращает их произведение $x \cdot y$. Ее можно реализовать посредством многократного (y раз) добавления x к локальной переменной, допустим, `sum`, а затем возвращения значения `sum`. Функция, реализующая этот примитивный алгоритм умножения, приведена на иллюстрации 8.1. Этот пример демонстрирует то, как типичную логику цикла можно выразить с помощью команд ветвления ВМ `goto`, `if-goto` и `label`.

Высокоуровневый код	Код ВМ
<pre>// Returns x * y int mult(int x, int y) { int sum \ 0; int i \ 0; while (i < y) { sum += x; i++; } return sum; }</pre>	<pre>// Returns x * y function mult(x,y) push 0 pop sum push 0 pop i label WHILE_LOOP push i push y lt neg if-goto WHILE_END push sum push x add pop sum push i push 1 add pop i goto WHILE_LOOP label WHILE_END push sum return</pre>

Иллюстрация 8.1. Действие команды ветвления. В коде ВМ справа для облегчения чтения используются символические имена переменных вместо сегментов виртуальной памяти.

Обратите внимание, что булево условие `!(i < y)`, реализованное как `push i, push y, lt, ng`, помещается в стек непосредственно перед командой `if-goto WHILE_END`. В главе 7 мы увидели, что команды ВМ можно использовать для передачи и оценки любого булева выражения. Как показано на иллюстрации 8.1, высокоуровневые структуры управления типа `if` и `while` можно легко реализовать с помощью команд `goto` и `if-goto`. В целом же нашим (довольно минимальным) набором логических команд и команд ветвления ВМ можно реализовать любую структуру управления потоком высокоуровневого языка программирования.

Реализация: большинство низкоуровневых машинных языков, включая Наск, имеют средства для объявления символьных меток и для выполнения условных и безусловных действий типа «`goto label`». Поэтому, если наша реализация ВМ будет основана на программе, транслирующей команды ВМ в ассемблерные инструкции, реализовать команды ветвления ВМ будет относительно просто.

Операционная система: закончим этот раздел двумя попутными замечаниями. Во-первых, программы ВМ не пишутся людьми. Их, как правило, пишут компиляторы. На иллюстрации 8.1 слева показан исходный код, а справа — код ВМ. В главах 10–11 мы разработаем *компилятор*, преобразующий первый во второй. Во-вторых, обратите внимание на низкую эффективность реализации функции `mult`. Позже в книге мы представим оптимизированные алгоритмы умножения и деления, работающие на уровне битов. Эти алгоритмы будут использоваться для реализации функций `Math.multiply` и `Math.divide`, входящих в состав операционной системы, которую мы построим в главе 12.

Наша ОС будет написана на языке Jack и переведена компилятором Jack на язык ВМ. В результате получится библиотека из восьми файлов: `Math.vm`, `Memory.vm`, `String.vm`, `Array.vm`, `Output.vm`, `Screen.vm`, `Keyboard.vm` и `Sys.vm` (API ОС приведен в приложении 6). Каждый файл ОС содержит набор полезных функций, которые может вызвать любая функция ВМ в процессе своей работы. Например, всякий раз, когда функции ВМ потребуется выполнить умножение

или деление, она сможет вызвать функцию `Math.multiply` или `Math.divide`.

8.3. Функции

Каждый язык программирования характеризуется фиксированным набором встроенных операций. Кроме того, языки высокого уровня и некоторые низкоуровневые языки предлагают большую свободу расширения этого фиксированного репертуара за счет неограниченного набора операций, которые может определять сам программист. В зависимости от языка, эти специально составленные операции обычно называются *подпрограммами*, *процедурами*, *методами* или *функциями*. В нашем языке ВМ все эти единицы программирования будут называться *функциями*.

В хорошо продуманных языках встроенные команды и те функции, которые формулирует программист, похожи друг на друга по своему виду и поведению. Например, чтобы вычислить $x + y$ на нашей стековой машине, мы выполняем следующие команды: `push x, push y, add`. При этом ожидаем, что реализация `add` извлечет два верхних значения из стека, сложит их и поместит результат в стек. Предположим теперь, что либо мы, либо кто-то другой написал функцию `power`, предназначенную для возведения числа в степень, то есть вычисления x^y . Чтобы воспользоваться этой функцией, мы выполняем похожий ряд программ: `push x, push y, call power`. Такая последовательность протокола позволяет совершенно естественным образом сочетать примитивные команды с вызовами функций. Например, выражение $(x + y)^3$ можно вычислить с помощью команд `push x, push y, add, push 3, call power`.

Как видно, единственное различие между использованием примитивной операции и вызовом функции заключается в ключевом слове `call`, предшествующем функции. Все остальное абсолютно одинаково: обе операции требуют, чтобы вызывающая функция «подготовила сцену», поместив аргументы в стек, обе операции должны использовать и удалить свои аргументы, и обе операции должны поместить возвращаемые

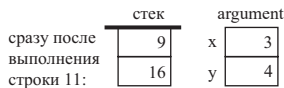
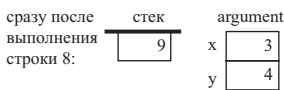
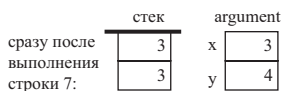
значения в стек. Этому протоколу вызова присуща элегантная последовательность, красота которой, надеемся, не ускользнет от читателя.

```
0 function main()
// Вычисляет hypot(3,4)
1  push 3
2  push 4
3  call hypot
4  return

5 function hypot(x,y)
// Вычисляет sqrt(x*x + y*y)
6  push x
7  push x
8  call mult
9  push y
10 push y
11 call mult
12 add
13 call sqrt
14 return

15 function mult(x,y)
// Вычисляет x * y (как на иллюстрации 8.1)
16 push 0
17 pop sum
18 push 0
19 pop i
...
36 push sum
37 return
```

Мир функции hypot,
имеющей 2 аргумента и без локальных переменных, показанный во время выполнения команды `call hypot(3,4)`:



Мир функции mult,
имеющей 2 аргумента и 2 локальные переменные, показанный во время выполнения команды `call mult(3,3)`:

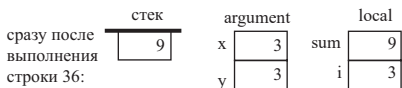
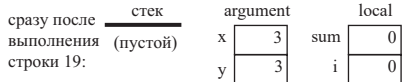


Иллюстрация 8.2. Снимки состояния стека и сегментов во время выполнения программы из трех функций. Номера строк не являются частью кода и приведены только для справки.

Пример: на иллюстрации 8.2 показана программа ВМ, вычисляющая функцию $\sqrt{x^2 + y^2}$, то есть длину гипотенузы прямоугольного треугольника — во многих программных библиотеках эта функция носит имя *hypot*. Программа состоит из трех функций со следующим поведением во время выполнения: *main* вызывает *hypot*, а затем *hypot* дважды вызывает *mult*. Имеется также вызов функции *sqrt*, которую мы не отслеживаем ради простоты анализа.

В нижней части иллюстрации 8.2 показано, что во время своего выполнения каждая функция видит свой частный «мир», состоящий из ее собственного рабочего стека и ее собственных сегментов памяти. Эти отдельные миры связаны между собой двумя «червоточинами»: когда функция говорит *call mult*, аргументы, которые она поместила в свой стек до вызова, каким-то образом передаются в сегмент *argument* вызываемой функции. Аналогично, когда функция говорит *return*, последнее значение, которое она поместила в свой стек непосредственно перед возвратом, каким-то образом копируется в стек вызывающей функции, заменяя ранее помещенные аргументы. Все эти действия по переносу аргументов и взаимодействию функций выполняются реализацией ВМ, о чем мы сейчас и расскажем.

Реализация: типичная компьютерная программа состоит, как правило, из нескольких, а возможно, и огромного множества функций. Однако в конкретный момент выполнения программы лишь некоторые из этих функций действительно что-то делают. Воспользуемся термином *цепочка вызовов* для концептуального обозначения всех функций, которые участвуют в выполнении программы в данный момент. Когда программа ВМ начинает выполняться, цепочка вызовов состоит только из одной функции, скажем, *main* («главная функция»). В какой-то момент *main* может вызвать другую функцию, скажем, *foo*, а эта функция, в свою очередь, может вызвать еще одну функцию, скажем, *bar*. В этот момент цепочка вызовов выглядит так: *main*→*foo*→*bar*. Каждая функция в цепочке вызовов ожидает возвращения функции, которую она вызвала. Таким образом, единственная функция, которая действительно активна в цепочке вызовов, — это последняя, которую мы назовем *текущей функцией*, то есть выполняемой в данный момент.

Для выполнения своей работы функции обычно используют *локальные переменные* и *переменные аргументов*. Эти переменные являются временными: сегменты памяти, которые их представляют, назначаются во время начала выполнения функции, и их можно повторно использовать после того, как функция вернет свое значение. Эта задача по управлению памятью осложняется тем, что вызов функций носит произвольный характер (функцию можно вызвать в любом месте и любой по счету), а также может быть рекурсивным (функция может вызвать себя). Во время выполнения программы каждый вызов функции должен обрабатываться независимо от всех других вызовов и поддерживать свой собственный стек, свои локальные переменные и свои переменные аргументов. Как же реализовать этот механизм неограниченной вложенности и связанные с ним задачи управления памятью?

Свойство, которое делает данную задачу по «ведению домашнего хозяйства» (реализации служебных задач) выполнимой, заключается в линейной природе логики вызова и возврата. Хотя цепочка вызовов функций может быть произвольно глубокой и рекурсивной, в любой момент времени в конце цепочки выполняется только одна функция, а все остальные функции в цепочке ожидают ее возвращения. Такая модель обработки *«последним пришел — первым вышел»* (*last-in-first-out*, LIFO) идеально подходит для стековой структуры данных, также основанной на модели LIFO. Рассмотрим это подробнее.

Предположим, что текущей функцией является `f00`. Предположим, что `f00` уже поместила некоторые значения в свой рабочий стек и изменила некоторые записи в своих сегментах памяти. Предположим, что в какой-то момент `f00` захотела вызвать другую функцию, `bar`, для получения некоего значения. В этот момент мы должны приостановить выполнение `f00` до тех пор, пока не завершит свое выполнение `bar`. «Заморозка» рабочего стека `f00` не представляет проблемы: поскольку стек растет только в одном направлении, рабочий стек `bar` никогда не перезапишет ранее загруженные значения. Поэтому сохранить рабочий стек вызывающей программы очень просто — задача выполняется сама по себе благодаря линейной и односторонней структуре стека. Но как сохранить сегменты памяти `f00`? Вспомним, что в главе 7 мы использовали указатели `LCL`, `ARG`, `THIS`

и THAT для ссылки на базовые адреса оперативной памяти сегментов local, argument, this и that текущей функции. Если мы хотим «заморозить» эти сегменты, то можем поместить их указатели в стек и извлечь позже, когда захотим вернуть к жизни foo. В дальнейшем для обозначения совокупности значений указателей, необходимых для сохранения и восстановления состояния функции, мы будем использовать термин *стековый кадр*, или *фрейм*.

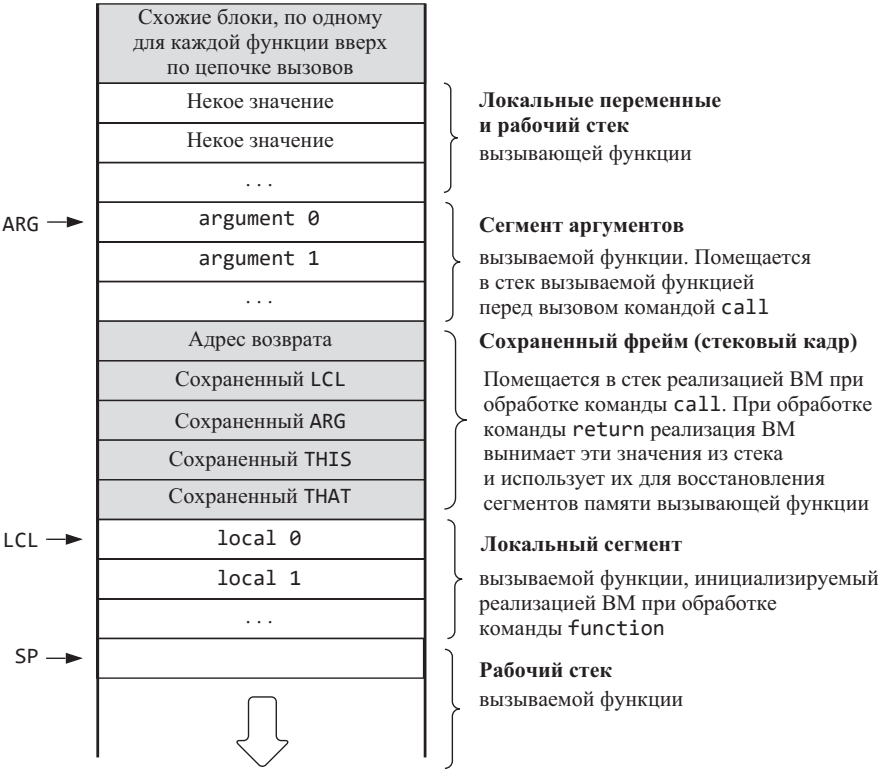


Иллюстрация 8.3. Глобальный стек, показанный во время работы вызываемой процедуры. Перед завершением работы вызываемая процедура помещает в стек возвращаемое значение (не показано). Обработывая команду возврата, реализация ВМ копирует значение возврата в argument 0 и устанавливает SP на адрес, следующий сразу за ним. Это эффективно освобождает область глобального стека под новым значением SP. Таким образом, когда вызывающая процедура возобновляет выполнение, она видит возвращаемое значение на вершине своего рабочего стека.

Итак, как видно, при переходе из однофункциональной среды к многофункциональной, казалось бы, скромный стек начинает играть весьма существенную роль в нашей истории. В частности, теперь мы используем одну и ту же структуру данных для хранения как рабочих стеков, так и фреймов всех функций в цепочке вызовов. Чтобы оказать ей заслуженное уважение, отныне мы будем называть эту трудолюбивую структуру данных *глобальным стеком*. Подробности показаны на иллюстрации 8.3.

Как показано на иллюстрации 8.3, при обработке команды `call functionName` реализация ВМ помещает в стек фрейм вызывающей функции. По окончании этой подготовительной задачи мы готовы перейти к выполнению кода вызываемой функции. Реализовать такой мега-переход несложно. Как мы увидим позже, при обработке команды `function functionName` имя функции используется для создания и включения в генерируемый поток ассемблерного кода уникальной символической метки на начало функции. Таким образом, обрабатывая команду типа «`function functionName`», мы можем сгенерировать ассемблерный код, выполняющий по сути операцию «`goto functionName`». При выполнении эта команда эффективно передаст управление вызываемой функции.

Возврат от вызываемой функции к вызывающей, когда первая завершает работу, сложнее, потому что команда ВМ `return` не указывает адрес возврата. И действительно, отсутствие адреса возврата подразумевается в самой концепции вызова функции: такие функции, как `mult` или `sqrt`, предназначены для обслуживания любой вызывающей функции, так что задать адрес возврата невозможно по определению. Поэтому команда `return` интерпретируется следующим образом: перенаправить выполнение программы на участок памяти, содержащий команду, следующую сразу за командой `call`, вызвавшей текущую функцию.

Реализация виртуальной машины может выполнить это соглашение посредством: 1) сохранения адреса возврата непосредственно перед передачей управления и 2) получения адреса возврата и перехода к нему сразу после возвращения значения вызывающей функцией. Но где же хранить адрес возврата? И снова на помощь приходит

наш трудолюбивый стек. Напомним, что транслятор ВМ переходит от одной команды ВМ к другой, генерируя код по ходу дела. Встречая в коде ВМ команду `call foo`, мы точно знаем, какая команда должна быть выполнена при завершении `foo`: это ассемблерная команда, следующая сразу за ассемблерными командами, реализующими команду `call foo`. Таким образом, мы можем попросить транслятор ВМ подставить метку прямо здесь, в сгенерированный поток ассемблерного кода, и поместить эту метку в стек. Позже, встретив в программе ВМ команду `return`, мы можем вынуть из стека ранее сохраненный адрес возврата — назовем его *returnAddress* — и выполнить операцию `goto returnAddress` на ассемблере. Это низкоуровневый трюк, который позволяет использовать магию среды исполнения для передачи управления обратно в нужное место в коде вызывающей функции.

Реализация ВМ в действии: перейдем теперь к пошаговой демонстрации того, как реализация ВМ поддерживает действие вызова и возврата функции. Рассмотрим ее работу в контексте выполнения функции `factorial`, предназначенной для рекурсивного вычисления $n!$. На иллюстрации 8.4 показан код вместе с фрагментами состояния глобального стека во время выполнения `factorial(3)`. Полное моделирование этого вычисления должно также включать вызовы и возвраты функции `mult`, которая в данном конкретном примере вызывается дважды: один раз перед возвратом `factorial(2)` и один раз перед возвратом `factorial(3)`.

Сосредоточимся на самой левой и самой правой нижних частях иллюстрации 8.4 и попробуем рассмотреть обстановку как бы с точки зрения функции `main`: «Чтобы подготовить сцену, я поместила константу 3 в стек, а затем вызвала `factorial` для вычисления (см. самый левый снимок стека). В этот момент я уснула; в какой-то момент времени меня разбудили, и я узнала, что стек теперь содержит 6 (см. последний и самый правый снимок стека); я понятия не имею, каким волшебным образом это произошло, и меня это не очень волнует; все, что я знаю, так это то, что я задалась целью вычислить $3!$ и получила именно то, что просила». Другими словами, вызывающая функция совершенно не заметила той мини-драмы, которую породила ее команда `call`.

Высокоуровневый код

```
// Тестирует функцию факториала
int main() {
    return factorial(3);
}

// Вычисляет n!
int factorial(int n) {
    if (n\\1)
        return 1;
    else
        return n * factorial(n-1);
}
```

Код ВМ

```
// Тестирует функцию факториала
function main
    push 3
    call factorial
    return

// Вычисляет n!
function factorial(n)
    push n
    push 1
    eq
    if-goto BASE_CASE
    push n
    push n
    push 1
    sub
    call factorial
    call mult
    return
label BASE_CASE
    push 1
    return
```

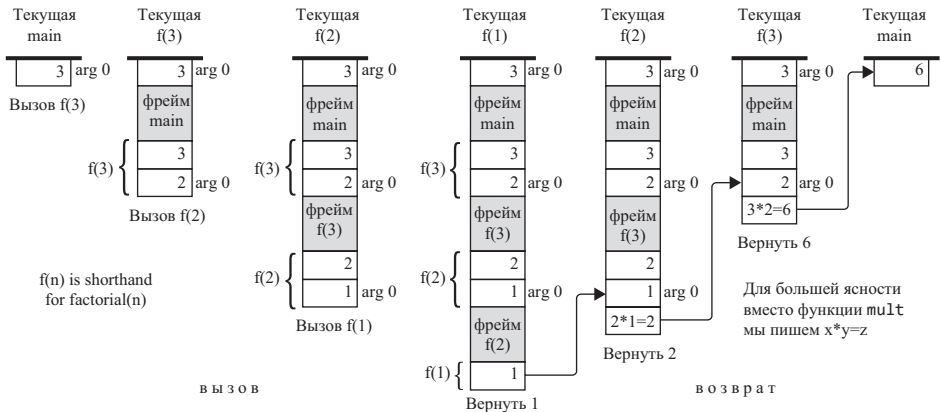


Иллюстрация 8.4. Несколько снимков состояния *глобального стека* во время выполнения функции `main`, вызывающей `factorial` для вычисления $3!$. Текущая функция видит только свой рабочий стек — область белого цвета на конце глобального стека; другие белые области глобального стека — рабочие стеки функций вверх по цепочке вызовов, ожидающих возвращения текущей функции. Обратите внимание на то, что серые области не соответствуют им «по масштабу», так как каждый фрейм состоит из пяти слов, как показано на иллюстрации 8.3.

Как показано на иллюстрации 8.4, сцена, на которой разыгрывается эта драма — это глобальный стек, а хореограф, руководящий постановкой, — реализация виртуальной машины: каждая операция вызова выполняется посредством сохранения в стеке фрейма (стекового кадра) вызывающей функции и перехода к исполнению вызываемой функции. Каждая операция возврата реализуется посредством: 1) использования последнего сохраненного фрейма для получения адреса возврата в коде вызывающей функции и восстановления ее сегментов памяти, 2) копирования самого верхнего значения стека (возвращаемого значения) на участок стека, связанный с аргументом `argument 0`, и 3) перехода к выполнению кода вызывающей функции, начиная с адреса возврата. Все эти операции должны реализовываться сгенерированным ассемблерным кодом.

Некоторые читатели могут удивиться, почему мы должны вдаваться во все эти подробности. На это есть по крайней мере три причины. Во-первых, подробности нужны нам для реализации транслятора ВМ. Во-вторых, реализация протокола вызова и возврата функций — прекрасный пример низкоуровневой программной инженерии, так что мы можем просто немного насладиться, наблюдая ее в действии. В-третьих, глубокое понимание принципов работы виртуальной машины поможет нам улучшить свои навыки, получить дополнительные знания и повысить свой уровень программиста. Работа со стеком, например, позволяет глубже понять преимущества рекурсии и связанные с ней подводные камни. Обратите внимание, что во время выполнения функции каждый рекурсивный вызов заставляет реализацию ВМ добавлять в стек блок памяти, состоящий из аргументов, фреймов функций, локальных переменных и рабочего стека вызываемой функции. Поэтому неконтролируемое использование рекурсии вполне может привести к печально известной проблеме *переполнения стека*. Это, а также соображения эффективности заставляют авторов компиляторов по мере возможностей преобразовывать рекурсивный код в последовательный. Но это уже совсем другая история, которую мы рассмотрим в главе 11.

8.4. Спецификация ВМ, часть II

До сих пор в этой главе мы описывали общие команды ВМ, не придерживаясь точного синтаксиса и соглашений. Теперь переходим к формальному определению команд *ветвления* ВМ, команд *функций* ВМ и структуры программ ВМ. На этом завершается спецификация языка ВМ, которую мы начали описывать в главе 7 «Спецификация ВМ, часть I».

Важно повторить, что обычно программы ВМ не пишутся людьми, а генерируются компиляторами. Поэтому описанные здесь спецификации предназначены для разработчиков компиляторов. То есть, если вы пишете компилятор, который должен транслировать программы с какого-либо языка высокого уровня в код ВМ, ожидается, что код, который сгенерирует ваш компилятор, будет соответствовать описанным здесь соглашениям.

Команды ветвления

- `label label`: отмечает текущее местоположение в коде функции. Переход возможен только к отмеченным местам. Область действия метки — функция, в которой она определена. Метка `label` — это строка, состоящая из любой последовательности букв, цифр, символа подчеркивания (`_`), точки (`.`) и двоеточия (`:`), которая не начинается с цифры. Команда `label` может располагаться в любом месте функции, до или после команд `goto`, которые ссылаются на нее.
- `goto label`: выполняет безусловную операцию `goto`, заставляя программу продолжить выполнение с отмеченного места. Команда `goto` и отмеченное место перехода должны находиться в одной и той же функции.
- `if-goto label`: выполняет условную операцию `goto`. Из стека берется самое верхнее значение; если значение не равно нулю, выполнение программы продолжается с отмеченного места; в противном случае выполнение программы продолжается со следующей команды. Команда `if-goto` и отмеченное место перехода должны находиться в одной и той же функции.

Команды функций

- `function functionName nVars`: отмечает начало функции с именем `functionName`. Команда сообщает, что функция имеет локальные переменные `nVars`.
- `call functionName nArgs`: вызывает именованную функцию. Команда сообщает, что перед вызовом в стек помещаются аргументы `nArgs`.
- `return`: передает выполнение команде, следующей сразу за командой `call` в коде вызывающей функции.

Программа VM

Программы VM генерируются из программ, написанных на языках высокого уровня типа Jack. Как мы увидим в следующей главе, программу на языке высокого уровня Jack в широком смысле можно определить как набор одного или нескольких файлов классов `.jack`, хранящихся в одной папке. Когда компилятор Jack применяется к этой папке, он переводит каждый файл класса `FileName.jack` в соответствующий файл `FileName.vm`, содержащий команды VM.

В ходе компиляции каждый конструктор, функция (статический метод) и метод с именем `bar` в файле `Jack FileName.jack` транслируются в соответствующую функцию VM с уникальным именем функции VM `FileName.bar`. Область видимости имен функций VM — глобальная: все функции VM во всех файлах `.vm` в папке программы видят друг друга и могут вызывать друг друга по уникальному и полному имени `FileName.functionName`.

Точка входа программы: один файл в любой программе Jack должен иметь имя `Main.jack`, и одна функция в этом файле должна иметь имя `main`. Таким образом, после компиляции один файл в любой программе VM должен иметь имя `Main.vm`, а одна функция VM в этом файле должна иметь имя `Main.main`, которая является точкой входа приложения. Это соглашение о среде выполнения реализуется следующим образом. Когда мы запускаем программу VM, первой всегда

выполняется функция ВМ без аргументов под названием `Sys.init`, являющаяся частью операционной системы. Эта функция ОС запрограммирована на вызов функции точки входа в пользовательской программе. В случае с `Jack Sys.init` запрограммирована на вызов `Main.main`.

Выполнение программы: существует несколько способов выполнения программ ВМ, один из которых — использование прилагаемого эмулятора ВМ, о котором говорилось в главе 7. При загрузке в эмулятор ВМ папки с программой, содержащей один или несколько файлов `.vm`, эмулятор загружает все функции ВМ во всех этих файлах одну за другой (порядок загрузки функций ВМ несущественен). В результате создается кодовая база, представляющая собой последовательность всех функций ВМ во всех файлах `.vm` в папке программы. Понятие файлов ВМ перестает существовать, хотя оно подразумевается в именах загруженных функций ВМ (*FileName.functionName*).

Эмулятор ВМ «От Nand до “Тетриса”», представляющий собой программу на Java, имеет встроенную реализацию ОС Jack, также написанную на Java. Обнаруживая вызов функции ОС, например `call Math.sqrt`, эмулятор поступает следующим образом. Если он находит соответствующую команду функции `Math.sqrt` в загруженном коде ВМ, то он выполняет код функции ВМ. В противном случае эмулятор использует свою встроенную реализацию метода `Math.sqrt`. Это означает, что, если для выполнения программ ВМ вы пользуетесь поставляемым эмулятором ВМ, необходимости включать файлы ОС в ваш код нет. Эмулятор ВМ будет обслуживать все встречающиеся в вашей программе вызовы ОС, используя свою встроенную реализацию ОС.

8.5. Реализация

В предыдущем разделе была завершена спецификация нашего языка и структуры ВМ. В этом разделе мы сосредоточимся на вопросах реализации, что поможет вам создать полностью рабочий транслятор с языка ВМ на язык ассемблера Nask. Раздел 8.5.1 содержит советы

по реализации протокола вызова и возврата функций. Раздел 8.5.2 завершает стандартное отображение реализации ВМ на платформу Nask. В разделе 8.5.3 описывается предполагаемый дизайн и API для завершения транслятора ВМ, который мы начали создавать в проекте 7.

8.5.1. Вызов и возврат функций

События вызова функции и возврата из вызова функции можно рассматривать с двух точек зрения: с точки зрения *вызывающей функции* и с точки зрения *вызываемой функции*. Как вызывающая, так и вызываемая функции имеют определенные ожидания и определенные обязанности в отношении обработки команд вызова, функции и возврата. Выполнение ожиданий одной — обязанность другой. Кроме того, важную роль в выполнении этого соглашения играет реализация виртуальной машины. Далее обязанности реализации ВМ обозначены символом [†].

Точка зрения вызывающей функции:	Точка зрения вызываемой функции:
• Перед вызовом функции я должна поместить в стек столько аргументов (<i>nArgs</i>), сколько ожидает получить вызывающая функция. • Далее я вызываю функцию с помощью команды <code>call fileName.functionName nArgs</code>. • После возврата помещенные мной в стек аргументы исчезают из стека, а возвращаемое значение (существующее всегда) появляется на вершине стека. За исключением этого изменения, мой рабочий стек остается таким же, каким был до вызова [†]. • После возврата все мои сегменты памяти остаются точно такими же, какими были до вызова [†], за исключением того, что содержимое моего сегмента <code>static</code> может измениться, а сегмент <code>temp</code> может быть не определен.	• Перед началом выполнения мой сегмент <code>argument</code> был инициализирован значениями аргументов, переданных вызывающей функцией; также для меня был выделен и инициализирован нулями сегмент переменных <code>local</code>. Мой сегмент <code>static</code> установлен на сегмент <code>static</code> файла ВМ, к которому я принадлежу, а мой рабочий стек пуст. Сегменты памяти <code>this</code>, <code>that</code>, <code>pointer</code> и <code>temp</code> при входе не определены [†]. • Перед возвратом я должна поместить возвращаемое значение в стек.

Команда ВМ	Ассемблерный (псевдо) код, генерируемый транслятором ВМ
call <i>f nArgs</i> (вызывает функцию <i>f</i> , сообщая, что перед вы- зовом в стек были поме- щены аргументы <i>nArgs</i>)	push <i>returnAddress</i> // генерирует метку и помещает ее в стек push LCL // сохраняет LCL вызывающей функции push ARG // сохраняет ARG вызывающей функции push THIS // сохраняет THIS вызывающей функции push THAT // сохраняет THAT вызывающей функции ARG=SP-5- <i>nArgs</i> // перемещает ARG LCL=SP // перемещает LCL goto <i>f</i> // передает контроль вызываемой функции (<i>returnAddress</i>) // вставляет в код метку адреса возврата
function <i>f nVars</i> (объявляет функцию <i>f</i> , сообщая, что функция имеет <i>nVars</i> локальных переменных)	(<i>f</i>) // вставляет в код метку входа функции Повторить <i>nVars</i> раз: // <i>nVars</i> = количество местных переменных push 0 // обнуляет местные переменные
return (завершает текущую функцию и возвращает управление вызывающей)	<i>frame</i> LCL // <i>frame</i> — временная переменная <i>retAddr</i> = *(<i>frame</i> -5) // присваивает временной переменной значение адреса возврата *ARG = pop() // перемещает значение возврата для вызывающей функции SP ARG+1 // перемещает SP для вызывающей THAT *(<i>frame</i> -1) // возвращает THAT для вызывающей THIS *(<i>frame</i> -2) // возвращает THIS для вызывающей ARG *(<i>frame</i> -3) // возвращает ARG для вызывающей LCL *(<i>frame</i> -4) // возвращает LCL для вызывающей goto <i>retAddr</i> // переход к адресу возврата

Иллюстрация 8.5. Реализация команд функций языка ВМ. Все действия, описанные справа, реализуются с помощью генерируемых ассемблерных инструкций Hack.

Реализация ВМ поддерживает это соглашение, манипулируя структурой глобального стека, описанной на иллюстрации 8.3. В частности, каждая функция, команда вызова и возврата в коде ВМ обрабатываются посредством генерации ассемблерного кода, который оперирует глобальным стеком следующим образом. Команда вызова `call` генерирует код, который сохраняет в стеке фрейм (стековый кадр) вызывающей функции и переходит к выполнению вызываемой функции. Команда `function` генерирует код, инициализирующий локальные

переменные вызывающей функции. Наконец, команда `return` генерирует код, копирующий возвращаемое значение на вершину рабочего стека вызывающей программы, восстанавливает сегментные указатели вызывающей программы и переходит к выполнению вызывающей программы, начиная с адреса возврата. Подробности см. на иллюстрации 8.5.

8.5.2. Стандартное отображение ВМ на платформе Hask, часть II

Разработчикам реализации ВМ на компьютере Hask рекомендуется следовать описанным здесь соглашениям. Эти соглашения дополняют рекомендации, приведенные в разделе 7.4.1 «Стандартное отображение ВМ на платформе Hask, часть I».

Стек: на платформе Hask участки с 0 по 15 в оперативной памяти RAM зарезервированы для указателей и виртуальных регистров, а места с 16 по 255 зарезервированы для статических переменных. Стек отображается на адреса памяти, начиная с 256. Для реализации такой привязки транслятор ВМ должен начать с генерации ассемблерного кода, устанавливающего указатель стека `SP` в ячейку памяти с номером 256. С этого момента, встречая в исходном коде ВМ такие команды, как `pop`, `push`, `add` и прочие, транслятор ВМ генерирует ассемблерный код, влияющий на эти операции и манипулирующий адресом, на который указывает `SP`, изменяя `SP` по мере необходимости. Эти действия объяснялись в главе 7 и были реализованы в проекте 7.

Специальные символы: при трансляции команд ВМ на язык ассемблера Hask транслятор ВМ имеет дело с двумя типами символов. Во-первых, он управляет предопределенными символами ассемблерного уровня, такими как `SP`, `LCL` и `ARG`. Во-вторых, генерирует и использует символьные метки для обозначения адресов возврата и точек входа функций. Для иллюстрации вернемся к программе `PointDemo`, представленной во введении к части II. Эта программа состоит из двух

файлов классов `Jack`, `Main.jack` (иллюстрация II.1) и `Point.jack` (иллюстрация II.2), хранящихся в папке с именем `PointDemo`. Будучи примененным к этой папке, компилятор `Jack` создает два файла ВМ с именами `Main.vm` и `Point.vm`. Первый файл содержит единственную функцию ВМ `Main.main`, а второй содержит несколько функций ВМ: `Point.new`, `Point.getx`, ..., `Point.print`.

Когда к этой же папке применяется транслятор ВМ, он создает единственный файл ассемблерного кода с именем `PointDemo.asm`. На уровне ассемблерного кода абстракции функций уже не существуют. Вместо этого для каждой команды `function` транслятор ВМ генерирует метку входа в ассемблерном коде; для каждой команды `call` транслятор ВМ: 1) генерирует ассемблерную инструкцию `goto`; 2) создает метку адреса возврата и помещает ее в стек; 3) вставляет эту метку в сгенерированный код. Для каждой команды `return` транслятор ВМ извлекает адрес возврата из стека и генерирует инструкцию `goto`. Например:

<u>Код ВМ</u>	<u>Сгенерированный код ассемблера</u>
<code>function Main.main</code>	<code>(Main.main)</code>
<code>...</code>	<code>...</code>
<code>call Point.new</code>	<code>goto Point.new</code>
<code>// Следующая команда ВМ</code>	<code>(Main.main\$ret0)</code>
<code>...</code>	<code>// Следующая команда ВМ</code>
	<code>(в коде ассемблера)</code>
<code>function Point.new</code>	<code>...</code>
<code>...</code>	<code>(Point.new)</code>
<code>return</code>	<code>...</code>
	<code>goto Main.main\$ret0</code>

На иллюстрации 8.6 приведены спецификации для всех символов, обрабатываемых и генерируемых транслятором ВМ.

Символ	Использование
SP	Этот предопределенный символ указывает на адрес участка оперативной памяти хоста, следующий сразу после адреса участка памяти, содержащего самое верхнее значение стека
LCL, ARG, THIS, THAT	Эти предопределенные символы указывают на базовые адреса оперативной памяти виртуальных сегментов <code>local</code> , <code>argument</code> , <code>this</code> и <code>that</code> текущей запущенной функции ВМ
Символы <i>Xxx.i</i> (представляют статические переменные)	Каждая ссылка на <code>static i</code> , появляющаяся в файле <code>Xxx.vm</code> , транслируется в ассемблерный символ <code>Xxx.i</code> . В последующем процессе ассемблер Hack сопоставит эти символические переменные с участками оперативной памяти, начиная с адреса 16
<i>functionName \$label</i> (места назначения команд <code>goto</code>)	Пусть <code>foo</code> — это функция в файле <code>Xxx.vm</code> . При обработке каждой команды <code>label bar</code> внутри <code>foo</code> генерируется и вводится в поток кода ассемблера символ <code>Xxx.foo\$bar</code> . При трансляции команд <code>goto bar</code> и <code>if-goto bar</code> (внутри <code>foo</code>) в ассемблерные коды вместо <code>bar</code> должна использоваться метка <code>Xxx.foo\$bar</code>
<i>functionName</i> (символы точки входа функции)	При обработке каждой команды <code>function foo</code> в файле <code>Xxx.vm</code> генерируется и вводится в поток ассемблерного кода символ <code>Xxx.foo</code> , обозначающий точку входа функции. В последующем процессе ассемблер преобразует этот символ в физический адрес, с которого начинается код функции
<i>functionName \$ret.i</i> (символы адреса возврата)	Пусть <code>foo</code> — функция в файле <code>Xxx.vm</code> . При обработке каждой команды <code>call</code> в коде <code>foo</code> генерируется и вводится в поток кода ассемблера символ <code>Xxx.foo\$ret.i</code> , где <i>i</i> — целое число (один такой символ генерируется для каждой команды вызова в <code>foo</code>). Этот символ используется для маркировки адреса возврата в коде вызывающей функции. В последующем процессе ассемблер переведет этот символ в физический адрес памяти команды, следующей непосредственно за командой вызова
R13 — R15	Эти предопределенные символы можно использовать для любых целей. Например, R13 — R15 могут пригодиться, если транслятор ВМ генерирует ассемблерный код, в котором необходимо использовать низкоуровневые переменные для временного хранения

Иллюстрация 8.6. Описанные выше соглашения об именовании предназначены для того, чтобы в результате трансляции нескольких файлов и функций `.vm` в один файл `.asm` сгенерированные символы ассемблерного кода в пределах файла были уникальными.

Загрузочный код: стандартное отображение ВМ на платформу Hack предусматривает, что стек должен отображаться в оперативной памяти хоста, начиная с адреса 256, и что первой функцией ВМ должна выполняться функция ОС `Sys.init`. Как же реализовать эти соглашения на платформе Hack? Вспомните, что во время сборки компьютера Hack в главе 5 мы собрали его так, что при перезагрузке он получает и выполняет инструкцию, расположенную в ПЗУ по адресу 0. Таким образом, если нужно, чтобы при загрузке компьютер выполнял заранее определенный сегмент кода, можно поместить этот код в память команд компьютера Hack, начиная с адреса 0. Вот этот код:

```
// Загрузочный (псевдо) код, должен выражаться
// на машинном языке
SP = 256
call Sys.init
```

Затем функция `Sys.init`, являющаяся частью операционной системы, должна вызвать главную функцию приложения и войти в бесконечный цикл. Это действие заставит выполняться транслированную программу ВМ. Обратите внимание, что понятия «приложение» и «главная функция» в языках высокого уровня варьируются. В языке Jack принято, что `Sys.init` должна вызывать функцию ВМ `Main.main`. Это похоже на среду Java: когда мы даем JVM команду выполнить заданный класс Java, допустим, `Foo`, она ищет и выполняет метод `Foo.main`. В целом можно реализовать специфические для конкретного языка процедуры запуска, используя различные версии функции `Sys.init`.

Использование: транслятор принимает один аргумент командной строки следующим образом:

```
prompt> VMTranslator source,
```

где *source* — либо имя файла вида *Xxx.vm* (расширение обязательно), либо имя папки (в этом случае расширение отсутствует), содержащей

один или несколько файлов `.vm`. Имя файла/папки может содержать путь к файлу/папке. Если путь не указан, транслятор работает с текущей папкой. Результатом работы транслятора ВМ служит один ассемблерный файл с именем `source.asm`. Если источник — это имя папки, то единственный файл `.asm` содержит трансляцию всех функций во всех файлах `.vm` в папке, одной за другой. Выходной файл создается в той же папке, что и входной файл. Если в папке уже есть файл с таким именем, он будет перезаписан.

8.5.3. Предложения по реализации ВМ

В проекте 7 мы предлагали вам построить базовый транслятор ВМ с тремя модулями: `VMTranslator`, `Parser` и `CodeWriter`. Теперь мы опишем, как расширить эту базовую реализацию до полномасштабного транслятора ВМ. Сделать это можно, добавив к уже построенным в проекте 7 модулям описанную ниже функциональность. Нecessности разрабатывать дополнительные модули нет.

VMTranslator

Если входные данные для транслятора — это один файл, например `Prog.vm`, `VMTranslator` конструирует парсер `Parser` для разбора и составитель кода `CodeWriter`, который начинает работу с создания выходного файла с именем `Prog.asm`. Далее `VMTranslator` входит в цикл, который использует службы парсера для итерации входного файла и разбора каждой строки в виде команд ВМ за исключением пробелов. Для каждой разобранный команды `VMTranslator` использует `CodeWriter` для генерации ассемблерного кода `Hack` и записи сгенерированного кода в выходной файл. Все это уже было сделано в проекте 7.

Если на входе транслятору задается папка, названная, допустим, `Prog`, `VMTranslator` создает по парсеру для обработки каждого `.vm` файла в папке и один `CodeWriter` для генерации ассемблерного кода `Hack` в единственный выходной файл `Prog.asm`. Каждый раз, когда `VMTranslator` начинает транслировать новый файл `.vm`

в папке, он должен сообщить модулю CodeWriter, что сейчас обрабатывается новый файл. Для этого вызывается процедура модуля CodeWriter с именем `setFileName`, которую мы опишем ниже.

Parser

Этот модуль идентичен разработанному в проекте 7.

CodeWriter

CodeWriter, разработанный в проекте 7, предназначался для обработки *арифметико-логических* команд VM и команд *push/pop*. Вот API полного CodeWriter, обрабатывающего все команды языка VM.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Выходной файл / поток	—	Открывает выходной файл / поток и готовится к запи- си в него. Пишет инструк- ции для ассемблера, относя- щиеся к загрузочному коду, запускающему выполнение программы. Этот код нужно поместить в начало создавае- мого выходного файла / по- тока. Комментарий: См. «Со- веты по реализации» в конце раздела 8.6
<code>setFileName</code>	<code>filename</code> (строка)	—	Информирует о том, что на- чалась трансляция ново- го файла VM (вызывается <code>VMTranslator</code>)
<code>writeArtihmetic</code> (разработана в про- екте 7)	<code>command</code> (строка)	—	Записывает в выходной файл ассемблерный код, реализую- щий заданную арифметико- логическую команду <code>command</code>

writePushPop (разработана в проекте 7)	command (C_PUSH или C_POP), segment (строка), index (int)	—	Записывает в выходной файл ассемблерный код, реализующий заданную команду push или pop
writeLabel	label (строка)	—	Записывает ассемблерный код, реализующий команду label
writeGoto	label (строка)	—	Записывает ассемблерный код, реализующий команду goto
writeIf	label (строка)	—	Записывает ассемблерный код, реализующий команду if-goto
writeFunction	functionName (строка) nArgs (int)	—	Записывает ассемблерный код, реализующий команду function
writeCall	—	—	Записывает ассемблерный код, реализующий команду call
writeReturn	—	—	Записывает ассемблерный код, реализующий команду return
Close (разработана в проекте 7)		—	Закрывает выходной файл / поток

8.6. Проект

Говоря вкратце, мы должны расширить базовый транслятор, разработанный в главе 7, возможностями работы с несколькими файлами. `vm` и трансляции команд *ветвления* и команд *функций* ВМ в ассемблерный код Nask. На основе каждой разобранных парсером команды ВМ транслятор ВМ должен сгенерировать ассемблерный код, реализующий семантику команды на хосте-платформе Nask. Трансляция трех команд ветвления на ассемблер не представляет сложности. Трансляция трех команд функций — более сложная задача, предполагающая реализацию псевдокода, показанного на иллюстрации 8.5,

с использованием символов, описанных на иллюстрации 8.6. Мы повторяем совет, данный в предыдущей главе: начните с составления ассемблерного кода на бумаге. Нарисуйте изображения оперативной памяти и глобального стека, следите за указателем стека и указателями соответствующих сегментов памяти и убедитесь, что ваш бумажный ассемблерный код успешно реализует все низкоуровневые действия, связанные с обработкой команд вызова, функции и возврата.

Задача: расширить базовый транслятор VM, разработанный в проекте 7, до полномасштабного транслятора VM, предназначенного для обработки многофайловых программ, написанных на языке VM. Для этой версии транслятора VM предполагается, что исходный код VM не содержит ошибок. Проверку и обработку ошибок можно будет добавить в последующие версии транслятора VM, но в проект 8 они не входят.

Контракт: завершить создание транслятора с языка VM на язык ассемблера Nask, соответствующего спецификации VM, часть II (раздел 8.4) и стандартному отображению VM на платформу Nask, часть II (раздел 8.5.2). Воспользуйтесь своим транслятором для перевода предоставленных тестовых программ VM и получите соответствующие им программы, написанные на языке ассемблера Nask. При выполнении на прилагаемом эмуляторе ЦПУ сгенерированные вашим транслятором программы на языке ассемблера должны обеспечивать результаты, предусмотренные прилагаемыми тестовыми сценариями и файлами сравнения.

Ресурсы: вам понадобятся два инструмента: язык программирования, на котором вы будете реализовывать транслятор VM, и *эмулятор ЦПУ*, поставляемый в комплекте с программным обеспечением «От Nand до “Тетриса”». Воспользуйтесь эмулятором ЦПУ для выполнения и тестирования ассемблерного кода, сгенерированного вашим транслятором. Если сгенерированный код в эмуляторе ЦПУ работает правильно, будем считать, что ваш транслятор VM работает так, как ожидалось. Такого частичного тестирования наших целей будет достаточно.

Еще один инструмент, который пригодится в данном проекте, — поставляемый *эмулятор ВМ*. Воспользуйтесь этой программой для выполнения поставляемых тестовых программ ВМ и наблюдайте, как код ВМ влияет на смоделированные состояния стека и сегментов виртуальной памяти. Это поможет вам понять, какие действия в конечном счете должен реализовать транслятор ВМ в ассемблерном коде.

Поскольку полнофункциональный транслятор ВМ реализуется посредством расширения транслятора ВМ, собранного в проекте 7, вам также понадобится исходный код вашего базового транслятора.

Этапы тестирования и реализации

Мы рекомендуем выполнять реализацию транслятора ВМ в два этапа. Сначала реализуйте команды *ветвления*, а затем команды *функций*. Это позволит вам проводить модульное тестирование вашей реализации постепенно, с использованием прилагаемых тестовых программ.

Тестирование обработки команд ВМ **label, goto, if-goto**:

- **BasicLoop**: вычисляет $1 + 2 + \dots + \text{argument}[0]$ и помещает результат в стек. Проверяет, как транслятор ВМ обрабатывает команды **label** и **if-goto**.
- **FibonacciSeries**: вычисляет и сохраняет в памяти первые n элементов ряда Фибоначчи. Более строгий тест на обработку команд **label, goto** и **if-goto**.

Тестирование обработки ВМ **call, function, return**:

В отличие от проекта 7 теперь мы ожидаем, что транслятор ВМ будет обрабатывать многофайловые программы. Напомним, что по соглашению первой функцией в программе ВМ начинает выполняться `Sys.init`. Обычно `Sys.init` программируется для вызова функции `Main.main` программы. Однако для целей данного проекта мы используем поставляемые функции `Sys.init` для подготовки к различным тестам, которые хотим выполнить.

- `SimpleFunction`: выполняет простое вычисление и возвращает результат. Проверяет, как транслятор ВМ обрабатывает команды функции и возврата. Поскольку этот тест подразумевает обработку одного файла, состоящего из одной функции, тестовая функция `Sys.init` не нужна.
- `FibonacciElement`: эта тестовая программа состоит из двух файлов: `Main.vm` содержит единственную функцию `Fibonacci`, которая рекурсивно возвращает n -й элемент ряда Фибоначчи; `Sys.vm` содержит единственную функцию `Sys.init`, которая вызывает `Main.fibonacci` с $n = 4$ и затем входит в бесконечный цикл (напомним, что транслятор ВМ генерирует загрузочный код, вызывающий `Sys.init`). Полученная в результате рабочая среда обеспечивает строгую проверку работы транслятора ВМ с несколькими файлами `.vm`, командами вызова и возврата функций ВМ, загрузочным кодом и большинством других команд ВМ. Поскольку тестовая программа состоит из двух файлов `.vm`, нужно транслировать всю папку и создать один файл `FibonacciElement.asm`.
- `StaticsTest`: эта тестовая программа состоит из трех файлов: `Class1.vm` и `Class2.vm` содержат функции, которые устанавливают и получают значения нескольких статических переменных; `Sys.vm` содержит единственную функцию `Sys.init`, которая вызывает эти функции. Поскольку программа состоит из нескольких файлов `.vm`, нужно перевести всю папку и создать один файл `StaticsTest.asm`.

Советы по реализации

Поскольку проект 8 основан на расширении базового транслятора ВМ, разработанного в проекте 7, мы советуем сделать резервную копию исходного кода базового транслятора (если вы еще не сделали этого).

Начните с разработки ассемблерного кода, необходимого для реализации логики команд ВМ `label`, `goto` и `if-goto`. Затем приступайте к реализации методов `writeLabel`, `writeGoto` и `writeIf` модуля `CodeWriter`. Протестируйте созданный вами транслятор

ВМ, транслировав предоставленные программы `BasicLoop.vm` и `FibonacciSeries.vm`.

Загрузочный код: для того чтобы любая транслированная программа ВМ запустилась, в ней должен присутствовать код запуска, заставляющий реализацию ВМ начать выполнение программы на хост-платформе. Кроме того, для правильной работы любого кода ВМ реализация ВМ должна хранить базовые адреса стека и виртуальных сегментов в правильных участках оперативной памяти хоста. Первые три тестовые программы в этом проекте (`BasicLoop`, `FibonacciSeries`, `SimpleFunction`) предполагают, что код запуска еще не реализован, и включают тестовые скрипты, которые выполняют необходимые инициализации вручную, поэтому на данном этапе разработки вам не нужно беспокоиться об этом. Последние две тестовые программы (`FibonacciElement` и `StaticsTest`) предполагают, что код запуска уже входит в реализацию ВМ.

Как следствие, конструктор `CodeWriter` следует разрабатывать в два этапа. Первая версия вашего конструктора не должна генерировать никакого загрузочного кода (то есть игнорируйте руководство API конструктора, начинающееся с текста: «Пишет инструкции для ассемблера...»). Воспользуйтесь этой версией транслятора для модульного тестирования программ `BasicLoop`, `FibonacciSeries` и `SimpleFunction`. Вторая и последняя версия вашего конструктора `CodeWriter` должна записывать загрузочный код, как указано в API конструктора. Эту версию нужно использовать для модульного тестирования программ `FibonacciElement` и `StaticsTest`.

Поставляемые тестовые программы были тщательно спланированы для проверки специфических особенностей каждого этапа вашей реализации ВМ. Мы рекомендуем вам разрабатывать свой транслятор в предложенном порядке и тестировать его на каждом этапе с помощью соответствующих тестовых программ. Реализация более позднего этапа перед ранним может привести к сбою работы тестовых программ.

Веб-версия проекта 8 доступна на сайте www.nand2tetris.org.

8.7. Перспектива

Понятия *ветвления* и *вызова функций* являются фундаментальными для всех языков высокого уровня. Это означает, что в процессе перевода программ, написанных на высокоуровневых языках, в двоичный код кто-то должен позаботиться о выполнении сложных служебных задач, связанных с их реализацией. В Java, C#, Python и Jack это бремя ложится на уровень виртуальной машины. И для этого, как мы видели на протяжении всей этой главы, прекрасно подходит *стековая* архитектура виртуальной машины.

Чтобы оценить превосходные возможности нашей стековой модели ВМ, взгляните еще раз на представленные в этой главе программы. Например, на иллюстрациях 8.1 и 8.4 представлены высокоуровневые программы и их трансляции на язык ВМ. Если подсчитать строки, то можно заметить, что каждая строка высокоуровневого кода генерирует в среднем около четырех строк скомпилированного кода ВМ. Как выяснилось, при компиляции Jack-программ в код ВМ действительно поддерживается примерное соотношение 1:4. Краткость и читабельность кода ВМ, сгенерированного компилятором, можно оценить, даже не обладая особыми познаниями в сфере компиляции. Например, как мы увидим при создании компилятора, высокоуровневое высказывание типа `let y = Math.sqrt(x)` переводится в набор команд `push x, call Math.sqrt, pop y`. Двухуровневый компилятор может обойтись таким малым количеством работы, поскольку полагается на то, что остальную часть трансляции обработает реализация ВМ. Если бы нам пришлось переводить высокоуровневые высказывания типа `let y = Math.sqrt(x)` непосредственно на язык ассемблера Hack без услуг посредника в виде ВМ, то получившийся в результате код был бы гораздо менее элегантным и более загадочным.

Тем не менее такой вариант был бы более эффективным. Не будем забывать, что код ВМ должен реализовываться на машинном языке — именно этому были посвящены проекты 7 и 8. Как правило, конечный машинный код, полученный в результате двухуровневого процесса

трансляции, длиннее и менее эффективен, чем код, полученный в результате прямой трансляции. Итак, что более предпочтительно: двухуровневая программа на Java, которая в конечном счете генерирует тысячу машинных инструкций, или эквивалентная ей одноуровневая программа на C++, которая генерирует семьсот инструкций? Прагматичный ответ заключается в том, что каждый язык программирования имеет свои плюсы и минусы, а каждое приложение имеет свои операционные требования.

Одно из достоинств двухуровневой модели заключается в том, что промежуточным кодом ВМ (например, байт-кодом Java) можно *управлять*, например, с помощью программ, тестирующих его на наличие вредоносного кода или отслеживающих его для моделирования бизнес-процессов и т. д. В целом для большинства приложений преимущества *управляемого* кода оправдывают снижение его производительности на уровне ВМ. Однако для высокопроизводительных программ, таких как операционные системы и встроенные приложения, необходимость генерировать плотный и эффективный код обычно влечет за собой необходимость использования C/C++, компилируемого непосредственно в машинный язык.

Для составителей компиляторов очевидное преимущество явного промежуточного языка ВМ заключается в том, что он упрощает задачи по написанию и поддержке компиляторов. Например, разработанная в этой главе реализация ВМ освобождает компилятор от важных задач по обработке низкоуровневой реализации протокола вызова и возврата функций. В целом промежуточный слой ВМ позволяет разделить сложную задачу создания компилятора высокого уровня на две гораздо более простые: создание компилятора для перевода с высокоуровневого языка на язык ВМ и создание транслятора для перевода с языка ВМ на низкий уровень. Поскольку этот транслятор, который можно назвать «бэкендом» компилятора, уже был разработан в проектах 7 и 8, можно считать, что примерно половина общей задачи компиляции уже решена. Вторая половина — разработка «фронтенда» компилятора — будет рассматриваться в главах 10 и 11.

Закончим эту главу очередным замечанием о достоинствах разделения абстракции и реализации — такова общая тема практического

курса «От Nand до “Тетриса”» и важнейший принцип построения систем, выходящий далеко за рамки компиляции программ. Вспомним, что можно задавать функции ВМ, обращающиеся к своим сегментам памяти, с помощью таких команд, как `push argument 2`, `pop local 1` и т. д., не имея при этом ни малейшего представления о том, как эти значения представлены и как они сохраняются и восстанавливаются во время выполнения функции. Обо всех этих технических подробностях позаботится реализация виртуальной машины. Такое полное разделение абстракции и реализации подразумевает, что разработчикам компиляторов, генерирующих код ВМ, не нужно беспокоиться о том, как в конечном счете будет выполняться сгенерированный ими код; у них, как вы скоро поймете, хватает и своих проблем.

Так что выше нос! Вы на полпути к написанию двухуровневого компилятора для высокоуровневого, объектно-ориентированного, Java-подобного языка программирования. Следующая глава посвящена описанию этого языка. Она заложит основу для глав 10 и 11, в которых мы завершим разработку компилятора. Мы уже почти видим, как в конце туннеля осыпаются кирпичики «Тетриса».

9. Высокоуровневый язык

Высоким мыслям подобает высокий язык.

— *Аристофан (427–386 до н. э.)*

Представленные до сих пор в этой книге язык ассемблера и язык ВМ — это низкоуровневые языки, а это значит, что они предназначены для управления машинами, а не для разработки приложений. В этой главе мы представляем язык высокого уровня под названием Jack, предназначенный для того, чтобы программисты могли писать высокоуровневые программы. Jack — это простой объектно-ориентированный язык. По своим основным возможностям и принципам он похож на такие распространенные языки, как Java и C++, только с более простым синтаксисом и без поддержки наследования. Несмотря на свою простоту, Jack — это язык общего назначения, который можно использовать для создания множества самых разных приложений. В частности, он хорошо подходит для создания интерактивных игр, таких как «Тетрис», «Змейка», «Pong», «Space Invaders» и других подобных им.

Знакомство с Jack знаменует собой начало конца нашего путешествия. В главах 10 и 11 мы напишем компилятор, переводящий программы Jack в код ВМ, а в главе 12 разработаем простую операционную систему для платформы Jack/Нack. На этом создание компьютера завершится. Учитывая это, важно сразу сказать, что цель данной главы — не превратить вас в программиста на Jack. Мы также не утверждаем, что Jack имеет какую-то особую значимость вне контекста

курса «От Nand до “Тетриса”». Мы скорее рассматриваем Jack как необходимые строительные леса для глав 10–12, в которых будем создавать компилятор и операционную систему, делающие существование Jack возможным.

Если у вас есть опыт работы с современным объектно-ориентированным языком программирования, то с Jack вы сразу почувствуете себя как дома. Поэтому мы начинаем главу с нескольких примеров типичных программ на языке Jack. Все эти программы можно скомпилировать с помощью поставляемого компилятора Jack, находящегося в папке `nand2tetris/tools`. Скомпилированный им код VM в неизменном виде может быть выполнен на любой реализации VM, включая поставляемый эмулятор VM. В качестве альтернативы можно транслировать скомпилированный код VM на машинный язык с помощью транслятора VM, разработанного в главах 7–8. Полученный ассемблерный код можно запустить на прилагаемом эмуляторе ЦПУ или транслировать далее в двоичный код и выполнить на аппаратной платформе, построенной в главах 1–5.

Jack — простой язык, и простота его имеет свою цель. Во-первых, выучить (и забыть) Jack можно примерно за один час. Во-вторых, язык Jack был тщательно разработан специально для того, чтобы хорошо соответствовать распространенным методам компиляции. В результате вы можете с относительной легкостью написать элегантный *компилятор Jack*, что мы и сделаем в главах 10 и 11. Другими словами, намеренно простая структура Jack призвана помочь раскрыть программную инфраструктуру таких современных языков, как Java и C#. Вместо того чтобы разбирать компиляторы и среды выполнения этих языков, мы считаем, что полезнее разработать компилятор и среду выполнения самостоятельно, сосредоточившись на наиболее важных идеях, лежащих в основе их построения. Это будет сделано позже, в последних трех главах книги. Сейчас же, фигурально выражаясь, достанем Jack из коробки и посмотрим, что он собой представляет.

9.1. Примеры

Jack по большей части не требует особых объяснений, поэтому мы отложим спецификацию языка до следующего раздела и начнем с примеров. Первый пример — это наш старый знакомый *Hello World*. Второй пример иллюстрирует принципы процедурного программирования и обработки массивов. Третий демонстрирует реализацию абстрактных типов данных в языке Jack. Четвертый пример показывает реализацию связанного списка с использованием возможностей языка по работе с объектами.

На протяжении всех примеров мы кратко обсуждаем различные объектно-ориентированные идиомы и широко используемые структуры данных, предполагая, что читатель имеет базовое знакомство с этими темами. Если нет, читайте дальше — вы многое поймете, и у вас все получится.

Пример 1. Hello World: программа, показанная на иллюстрации 9.1, демонстрирует несколько основных возможностей языка Jack. По соглашению выполнение скомпилированной программы на Jack всегда начинается с функции `Main.main`. Таким образом, каждая программа Jack должна включать как минимум один класс с именем `Main`, а этот класс должен включать как минимум одну функцию `Main.main`. Это соглашение показано на иллюстрации 9.1.

```
/** Печатает "Hello World". File name: Main.jack */
class Main {
    function void main() {
        do Output.printString("Hello World");
        do Output.println(); // переход к новой строке;
        return;              // команда return обязательна
    }
}
```

Иллюстрация 9.1. Программа *Hello World*, написанная на языке Jack.

Jack поставляется со *стандартной библиотекой классов*, полное описание API которой приведено в приложении 6. Эта программная библиотека, также называемая *Jack ОС*, расширяет базовый язык различными абстракциями и сервисами, такими как математические функции, обработка строк, управление памятью, графика и функции ввода/вывода. Две такие функции ОС вызываются программой *Hello World* и нужны для вывода на печать. Программа также демонстрирует форматы комментариев, поддерживаемые Jack.

Пример 2. Процедурное программирование и работа с массивами: Jack содержит типичные операторы для обработки присваивания и итерации. Программа, показанная на иллюстрации 9.2, демонстрирует эти возможности в контексте обработки массивов.

```
/** Вводит последовательность целых чисел и высчитывает их среднее. */
class Main {
    function void main() {
        var Array a; // Массивы Jack не типизированы
        var int length;
        var int i, sum;
        let i \ 0;
        let sum \ 0;
        let length \ Keyboard.readInt("Сколько чисел? ") ;
        let a \ Array.new(length); // Конструирует массив
        while (i < length) {
            let a[i] \ Keyboard.readInt("Введите число: ") ;
            let sum \ sum + a[i];
            let i \ i + 1;
        }
        do Output.printString("Среднее значение: ");
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}
```

Иллюстрация 9.2. Типичное процедурное программирование и простая обработка массивов, выполняющаяся с использованием классов ОС Array, Keyboard и Output.

В большинстве высокоуровневых языков программирования объявление массивов входит в базовый синтаксис языка. В Jack мы предпочли рассматривать массивы как экземпляры класса `Array`, который является частью ОС, расширяющей базовый язык. Это было сделано из прагматических соображений, так как такое решение упрощает построение компиляторов Jack.

Пример 3. Абстрактные типы данных: в каждом языке программирования имеется фиксированный набор примитивных типов данных, которых в языке Jack три: `int`, `char` и `boolean`. В объектно-ориентированных языках программисты при необходимости могут вводить новые типы, создавая классы, представляющие абстрактные типы данных. Предположим, например, что мы хотим наделить Jack способностью работать с рациональными (дробными) числами, такими как $2/3$ и $314159/100000$ без потери точности. Это можно сделать, разработав отдельный класс Jack, предназначенный для создания и обработки дробных объектов вида x/y , где x и y — целые числа. Далее этот класс может представлять абстракцию дробей в любой программе Jack, которой необходимо оперировать рациональными числами. Перейдем теперь к описанию использования и разработке класса `Fraction`. Этот пример иллюстрирует принципы типичного многоклассового, объектно-ориентированного программирования в Jack.

Использование классов: на иллюстрации 9.3а приведен каркас класса (набор сигнатур методов), определяющий некоторые сервисы, которые можно ожидать от абстракции дробей. Такая спецификация часто называется *прикладным программным интерфейсом* (*Application Program Interface, API*). Клиентский код клиента в нижней части иллюстрации показывает, как этот API можно использовать для создания и манипулирования объектами дробей.

Иллюстрация 9.3а передает важный принцип программной инженерии: пользователям абстракции (такой как `Fraction`) не нужно знать ничего о ее реализации. Все, что им нужно, — это *интерфейс* класса, то есть API. API сообщает, какую функциональность

предлагает класс и как использовать эту функциональность. Это все, что нужно знать клиенту.

```

/** Представляет тип Fraction и соответствующие операции (каркас класса) */
class Fraction {

    /** Конструирует (сокращенную) дробь и x and y */
    constructor Fraction new(int x, int y)

    /** Возвращает числитель этой дроби */
    method int getNumerator()

    /** Возвращает знаменатель этой дроби */
    method int getDenominator()

    /** Возвращает сумму этой и другой дробей */
    method Fraction plus(Fraction other)

    /** Печатает эту дробь в формате x/y */
    method void print()

    /** Удаляет эту дробь */
    method void dispose() {

    // Еще связанные с дробью методы:
    // minus, times, div, invert, etc.
    }
}

```

```

// Вычисляет и печатает сумму 2/3 и 1/5
class Main {
    function void main() {
        // Создает 3 переменные дроби (указатели на объекты Fraction)
        var Fraction a, b, c;
        let a \ Fraction.new(4,6); // a \ 2/3
        let b \ Fraction.new(1,5); // b \ 1/5

        // Складывает две дроби и печатает результат
        let c \ a.plus(b); // c \ a + b
        do c.print(); // Should print "13/15"
        return;
    }
}

```

Иллюстрация 9.3а. API типа `Fraction` (наверху) и пример класса `Jack`, в котором создаются и используются объекты `Fraction`.

Реализация классов: до сих пор мы рассматривали класс `Fraction` с клиентской точки зрения как некую абстракцию в духе «черного ящика». На иллюстрации 9.3б приведена одна из возможных реализаций этой абстракции.

Класс `Fraction` иллюстрирует несколько ключевых особенностей объектно-ориентированного программирования в `Jack`. *Поля* (field) задают свойства объекта (также называемые *переменными экземпляров*). *Конструкторы* (constructor) — это процедуры (подпрограммы), создающие новые объекты, а *методы* (method) — это процедуры, работающие с текущим объектом (на который ссылается ключевое слово `this`). *Функции* (function) — это процедуры уровня класса (также

называемые *статическими методами*), которые не работают с каким-то конкретным объектом. Класс `Fraction` также демонстрирует все типы операторов, доступные в языке Jack: `let`, `do`, `if`, `while` и `return`. Конечно, класс `Fraction` — лишь один из примеров неограниченного числа классов, которые можно создать в Jack для поддержки любых целей программирования, какие только можно представить.

```

/** Представляет тип Fraction и соответствующие операции. */
class Fraction {
    // Каждый объект Fraction имеет числитель и знаменатель
    field int numerator, denominator;
    /** Конструирует (сокращенную) дробь from x and y */
    constructor Fraction new(int x, int y) {
        let numerator \ x;
        let denominator \ y;
        do reduce(); // Сокращает эту дробь
        return this; // Возвращает ссылку на новый объект
    }
    // Сокращает эту дробь
    method void reduce() {
        var int g;
        let g \ Fraction.gcd(numerator, denominator);
        if (g > 1) {
            let numerator \ numerator / g;
            let denominator \ denominator / g;
        }
        return;
    }
    // Вычисляет наибольший общий делитель двух данных целых
    function int gcd(int a, int b) {
        // Применяет алгоритм Евклида
        var int r;
        while (~(b \ 0)) {
            let r \ a - (b * (a / b)); // r = remainder
            let a \ b;
            let b \ r;
        }
        return a;
    }
}
// Объявление класса Fraction продолжается справа сверху

```

```

/** Методы доступа */
method int getNumerator() {
    return numerator;
}
method int getDenominator() {
    return denominator;
}
/** Возвращает сумму этой и другой дробей */
method Fraction plus(Fraction other) {
    var int sum;
    let sum \ (numerator * other.getDenominator()) +
        (other.getNumerator() * denominator);
    return Fraction.new(sum, denominator *
        other.getDenominator());
}
/** Выводит на печать эту дробь в формате x/y */
method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
}
/** Удаляет эту дробь */
method void dispose() {
    // Освобождает память от этого объекта
    do Memory.deAlloc(this);
    return;
}
// Больше относящихся к дробям методов можно объявить ниже:
// minus, times, div, invert, etc.
} // Конец декларации класса Fraction.

```

Иллюстрация 9.36. Реализация абстракции `Fraction` на языке Jack.

Пример 4. Реализация связанного списка: структура данных `list` определяется рекурсивно как значение, за которым следует список. Значение `null` — базовый случай определения — также считается списком. На иллюстрации 9.4 показана возможная Jack-реализация списка целых чисел. На этом примере видно, как можно использовать Jack для реализации одной из самых широко используемых в информатике структур данных.

```
/** Представляет список целых чисел. */
class List {
    field int data;      // Список состоит из целого значения,
    field List next;     // за которым следует List

    /* Создает список с головным элементом car и следующим
       элементом cdr */
    // Такие идентификаторы используются в памяти о языке Lisp
    constructor List new(int car, List cdr) {
        let data \ car;
        let next \ cdr;
        return this;
    }

    /* Методы доступа */
    method int getData() {return data;}
    method List getNext() {return next;}

    /* Вывод на печать элементов этого списка list */
    method void print() {
        // Инициализирует указатель на первом элементе этого списка
        var List current;
        let current \ this;
        // проходит по списку
        while (~(current \ null)) {
            do Output.printInt(current.getData());
            do Output.printChar(32); // Prints a space
            let current \ current.getNext();
        }
        return;
    }
}
// Декларация списка List продолжается справа сверху
```

```
/* Удаляет этот список*/
method void dispose() {
    // удаляет последующий элемент этого
    // списка, рекурсивно
    if (~(next \ null)) {
        do next.dispose();
    }
    // Использует рутину OS для освобождения
    // памяти, занимаемой этим объектом
    do Memory.deAlloc(this);
    return;
}

// Больше относящихся к спискам методов можно
// объявить ниже:
} // Конец декларации класса List.
```

```
// Пример клиентского кода:
// Создает, выводит на печать и удаляет
// список list (2,3,5), что служит сокращением
// для list (2,(3,(5, null))).
// (Этот код может появиться в любом классе Jack):
...
var List v;
let v \ List.new(5,null);
let v \ List.new(2,List.new(3,v));
do v.print();           // Печатает 2 3 5
do v.dispose();         // Удаляет список
...
```

Иллюстрация 9.4. Реализация связанного списка в Jack (слева и вверху справа) и пример использования (внизу справа).

Класс ОС	Службы
Math	Общие математические операции: max (int, int), sqrt (int)...
String	Представляет строки и связанные с ними операции: length (), charAt (int) ...
Array	Представляет массивы и связанные с ними операции: new (int), dispose ()
Output	Осуществляет вывод текста на экран: printString (String), printInt (int), println () ...
Screen	Осуществляет вывод графики на экран: setColor (Boolean), drawPixel (int, int), drawLine (int, int, int, int)...
Keyboard	Осуществляет ввод с клавиатуры
Memory	Осуществляет доступ к ОЗУ хоста: readLine (String), readInt (String) ...
Sys	Выполняет службы запуска: halt (), wait (int) ...

Иллюстрация 9.5. Службы операционной системы (краткое изложение). Полный API приведен в приложении 6.

Операционная система: Программы Jack широко используют операционную систему Jack, о которой будет идти речь в главе 12. Пока достаточно сказать, что программы Jack пользуются ОС абстрактно, без учета лежащей в ее основе реализации. Программы Jack могут использовать службы ОС напрямую, так что нет необходимости включать или импортировать какой-либо внешний код.

ОС состоит из восьми классов, кратко описанных на иллюстрации 9.5 и документированных в приложении 6.

9.2. Спецификация языка Jack

Этот раздел можно прочитать один раз, а затем при необходимости пользоваться им в качестве справочника.

9.2.1. Синтаксические элементы

Программа на языке Jack представляет собой последовательность токенов, разделенных произвольным количеством пробелов и комментариев. Токены могут быть символами, зарезервированными словами, константами и идентификаторами, как показано на иллюстрации 9.6.

9.2.2. Структура программ

Программа на языке Jack — это набор из одного или нескольких классов, хранящихся в одной папке. Один класс должен иметь имя `Main`, и этот класс должен содержать функцию с именем `main`. Выполнение скомпилированной программы Jack всегда начинается с функции `Main.main`.

Основная единица программирования в Jack — это *класс*. Каждый класс `Xxx` находится в отдельном файле с именем `Xxx.jack` и компилируется отдельно. По соглашению имена классов начинаются с заглавной буквы. Имя файла должно быть идентично имени класса,

включая регистр букв. Объявление класса имеет следующую структуру:

```
class имя {
    объявление переменной поля           //должно предшествовать
                                           объявлением процедуры
    объявление статической переменной //должно предшествовать
                                           объявлением процедуры
    объявление процедуры                 //объявления конструктора,
                                           метода и функции в любом
                                           порядке
}
```

Каждое объявление класса определяет имя, через которое можно получить глобальный доступ к сервисам класса. Далее следует последовательность из нуля или более объявлений полей и нуля или более объявлений статических переменных. Далее следует последовательность из одного или нескольких объявлений процедур, каждая из которых определяет *метод*, *функцию* или *конструктор*.

Методы предназначены для работы с текущим объектом. *Функции* — это статические методы на уровне класса, не связанные с каким-либо конкретным объектом. *Конструкторы* создают и возвращают новые объекты типа класса. Объявление процедуры имеет следующую структуру:

```
процедура тип имя (список параметров) {
    объявление местных переменных
    высказывания (операторы)
},
```

где *процедура* — это `constructor`, `method` или `function`. Каждая процедура имеет имя, через которое к ней можно обратиться, и *тип*, определяющий тип данных значения, возвращаемого процедурой. Если процедура не должна возвращать никакого значения, ее тип объявляется `void`. Список параметров представляет собой разделенный запятыми список пар *<тип идентификатор>*, например, `(int x, boolean sign, Fraction g)`.

Пробелы и комментарии	Символы пробела, новой строки и комментарии игнорируются. Поддерживаются следующие форматы комментариев: // Комментарий до конца строки /* Комментарий до закрытия */ /** Предназначен для программных инструментов, извлекающих документацию API */
Символы	() Используется для группировки арифметических выражений и формирования списков аргументов (в вызове процедур) и списков параметров (в объявлениях процедур) [] Используется для индексирования массивов { } Используется для группировки программных блоков и высказываний , Разделитель списка переменных ; Терминатор высказывания = Оператор присваивания и сравнения . Принадлежность к классу + - * / & ~ < > Операторы
Зарезервированные слова	class, constructor, method, function Компоненты программы int, boolean, char, void Примитивные типы let, do, if, else, while, return Высказывания true, false, null Значения-константы this Ссылка на объект
Константы	<i>Целочисленные константы</i> — значения в диапазоне от 0 до 32767. Отрицательные целые числа — не константы, а выражения, состоящие из унарного оператора минус, примененного к целочисленной константе. В результате допустимый диапазон значений составляет от -32768 до 32767 (первое значение можно получить с помощью выражения -32767 - 1) <i>Строковые константы</i> заключены в двойные кавычки («») и могут содержать любой символ, кроме символа новой строки или двойной кавычки. Эти символы предоставляются функциями OC String.newLine() и String.doubleQuote() <i>Булевы константы</i> — true и false Константа null означает ссылку на пустое значение
Идентификаторы	Идентификаторы составляются из произвольно длинных последовательностей букв (A-Z, a-z), цифр (0-9) и «_». Первый символ должен быть буквой или «_». Язык чувствителен к регистру: x и X рассматриваются как разные идентификаторы

Иллюстрация 9.6. Элементы синтаксиса языка Jack.

Если процедура — *метод* или *функция*, ее возвращаемый тип может быть любым из примитивных типов данных, поддерживаемых языком (*int*, *char* или *boolean*), любым из типов класса, предоставляемых стандартной библиотекой классов (*String* или *Array*), или любым из типов, реализуемых другими классами в программе (например, *Fraction* или *List*). Если процедура — *конструктор*, она может иметь произвольное имя, но ее тип должен быть именем класса, к которому она принадлежит. Класс может иметь 0, 1 или более конструкторов. По соглашению один из конструкторов называется *new*.

Следуя спецификации интерфейса, объявление процедуры содержит последовательность из нуля или более объявлений локальных переменных (операторов *var*), а затем последовательность из одного или более высказываний. Каждая процедура должна заканчиваться высказыванием *return выражение*. В случае процедуры типа *void*, когда возвращать нечего, процедура должна заканчиваться оператором *return* (его можно рассматривать как сокращение *return void*, где *void* — константа, обозначающая «ничего»). Конструкторы должны завершаться оператором *return this*. Это действие возвращает адрес памяти вновь созданного объекта, обозначаемого *this* (конструкторы Java делают то же самое, но неявно).

9.2.3. Типы данных

Переменные бывают следующих типов данных: либо это *примитивы* (*int*, *char* или *boolean*), либо *ClassName*, где *ClassName* — это *String*, *Array* или имя класса, находящегося в папке программы.

Примитивные типы: в Jack есть три примитивных типа данных:

int: 16-битное целое число с дополнительным кодом;
char: неотрицательное 16-битное целое число;
boolean: *true* или *false*.

Каждый из трех примитивных типов данных *int*, *char* и *boolean* внутренне представлен как 16-битное значение. Язык

слабо типизирован: значение любого типа может быть присвоено переменной любого типа без приведения.

Массивы: массивы объявляются с помощью класса `OSArray`. Доступ к элементам массива осуществляется с помощью типичной нотации `arr[i]`, где индекс первого элемента равен 0. Многомерный массив можно получить посредством создания массива массивов. Элементы массива не типизированы, и разные элементы одного и того же массива могут иметь разные типы. Объявление массива создает ссылку, а сам массив создается посредством выполнения вызова конструктора `Array.new(arrayLength)`. Пример работы с массивами приведен на иллюстрации 9.2.

Объектный тип: класс `Jack`, содержащий хотя бы один метод, определяет объектный тип. Как обычно в объектно-ориентированном программировании, создание объекта состоит из двух этапов. Вот пример:

(для большего понимания следует иметь в виду, что *car* — это «автомобиль», *employee* — «сотрудник», *name* — имя, а *license plate* — «регистрационный знак» — прим. пер.)

```
//Клиентский код с использованием классов Car
// и Employee, код которых здесь не показан.
// У класса Car два поля: model (String)
// и licensePlate (String).
// У класса Employee два поля: name (a String)
// и car (Car).
...
//Объявляет объект класса car и два объекта
// класса Employee (три указательные переменные):
var Car c;
var Employee emp1, emp2;
...
// Конструирует новый объект car:
let c = Car.new("Aston Martin","007"); // Присваивает
```



```
// переменной с базовый адрес блока памяти, содержащий
// данные нового car.

// Создает новый объект employee и назначает ему car:
let emp1 = Employee.new("Bond", c);
...
// Создает альтернативное имя для Bond:
let emp2 = emp1; // Копируется только ссылка (адрес),
// новый объект не создается.

// Теперь у нас есть два указателя Employee,
// ссылающиеся на один и тот же объект.
```

Строки: строки являются экземплярами класса OC String, реализующего строки как массивы значений char. Компилятор Jack распознает синтаксис "foo" и рассматривает его как объект String. Доступ к содержимому объекта String можно получить с помощью charAt(index) и других методов, описанных в API класса String (см. приложение 6). Вот пример:

```
var String s; // Объектная переменная
var char c; // Примитивная переменная
...
let s = "Hello World"; // Присваивает s значение
//объекта типа String "Hello World"
let c = s.charAt(6); // Присваивает c значение 87,
// то есть целочисленный символьный код 'W'
```

Высказывание `let s = "Hello World"` эквивалентно высказыванию `let s = String.new(11)`, за которым следуют одиннадцать вызовов методов `do s.appendChar(72), ..., do s.appendChar(100)`, где аргумент `appendChar` — целочисленный код символа. Фактически именно так компилятор и обрабатывает трансляцию `let s = "Hello World"`. Обратите внимание, что язык Jack не поддерживает идиому представления одного символа,

например, 'н'. Единственный способ представить символ — это использовать его целочисленный символьный код или вызвать метод `charAt`. Набор символов языка Haskell показан в приложении 5.

Преобразование типов: Haskell слабо типизирован: спецификация языка не определяет, что происходит, когда значение одного типа присваивается переменной другого типа. Решения о том, разрешать ли такие операции приведения и как их обрабатывать, остаются на усмотрение конкретных компиляторов Haskell. Спецификация специально оставлена неполной, что позволяет создавать минималистичные компиляторы, игнорирующие вопросы типизации. Тем не менее ожидается, что все компиляторы Haskell будут поддерживать и автоматически выполнять следующие присваивания.

- Символьное значение может быть присвоено целочисленной переменной, и наоборот, в соответствии со спецификацией набора символов Haskell (приложение 5). Пример:

```
var char c;  
let c = 33; // 'A'  
  
// Эквивалентно:  
var String s;  
let s = "A";  
let c = s.charAt(0);
```

- Целое значение может быть присвоено переменной-указателю (любого объектного типа), в этом случае оно интерпретируется как адрес памяти. Пример:

```
var Array arr; // Создает переменную-указатель  
let arr = 5000; // Присваивает arr значение 5000  
let arr[100] = 17; // Записывает значение 17 по адресу  
//памяти 5100
```

— Объектная переменная может быть присвоена переменной `Array`, и наоборот. Это позволяет обращаться к полям объекта как к элементам массива, и наоборот. Пример:

```
//Создает массив [2,5]:
var Array arr;
let arr = Array.new(2);
let arr[0] = 2;
let arr[1] = 5;

//Создает объект-дробь Fraction 2/5:
var Fraction x;
let x = arr; //Устанавливает для x базовый блок памяти,
             //представляющий массив [2, 5]
do Output.printInt(x.getNumerator()) //печатает "2"
do x.print()                        //печатает "2/5"
```

9.2.4. Переменные

В Jack есть четыре вида переменных. *Статические переменные* определяются на уровне класса и могут быть доступны всем процедурам класса. *Полевые переменные*, также определяемые на уровне класса, используются для представления свойств отдельных объектов и могут быть доступны всем конструкторам и методам класса. *Локальные переменные* используются процедурами для локальных вычислений, а *переменные параметров* представляют аргументы, которые были переданы процедуре вызывающей стороной. Локальные значения и значения параметров создаются непосредственно перед началом выполнения процедуры и утилизируются (удаляются) после ее возвращения. Подробности показаны на иллюстрации 9.7. *Область видимости* переменной — это область программы, в которой эта переменная распознается.

Вид	Описание	Где объявляются	Область видимости
Переменные класса	<code>static min varName1, varName2...</code> Существует одна копия каждой статической переменной, и эта копия совместно используется всеми процедурами класса (как <i>частные статические переменные</i> в Java)	Объявление класса	Класс, в котором объявлены
Полевые переменные	<code>Field min varName1, varName2...</code> Каждый объект (экземпляр класса) имеет частную копию переменных поля (как переменные-экземпляра в Java)	Объявление класса	Класс, в котором объявлены
Локальные переменные	<code>var min varName1, varName2...</code> Создаются при запуске процедуры и удаляются при ее возврате	Объявление процедуры	Процедура, в которой объявлены
Переменные параметров	Представляют собой аргументы, передаваемые процедуре. Рассматриваются как локальные переменные, значения которых инициализируются вызывающей процедурой	Объявление процедуры	Процедура, в которой объявлены

Иллюстрация 9.7. Виды переменных в языке Jack.

Инициализация переменных: статические переменные не инициализируются, и программист должен написать код, который инициализирует их перед использованием. Полевые переменные не инициализируются; предполагается, что их инициализируют конструктор или конструкторы класса. Локальные переменные не инициализируются, и программист сам решает, инициализировать ли их. Переменные параметров инициализируются значениями аргументов, передаваемых вызывающей функцией.

Видимость переменных: к статическим и полевым переменным нельзя получить прямой доступ вне класса, в котором они определены. Доступ к ним возможен только через методы доступа (аксессоры) и методы-модификаторы, как это предусмотрено создателем класса.

9.2.5. Высказывания

В языке Jack имеются пять высказываний, как показано на иллюстрации 9.8.

9.2.6. Выражения

Выражение в языке Jack — это одно из следующих:

- *Константа*.
- *Имя переменной* в области видимости. Переменная может быть *статической, полевой, локальной* или *переменной параметра*.
- Ключевое слово `this`, обозначающее текущий объект (не может использоваться в функциях).
- *Элемент массива*, в синтаксическом виде `arr[expression]`, где *arr* — имя переменной типа `Array` в области видимости.
- *Вызов процедуры*, возвращающий непустой тип.
- Выражение с префиксом в виде одного из унарных операторов — или `~`:
 - *выражение*: арифметическое отрицание;
 - *~выражение*: логическое отрицание (побитовое для целых чисел).
- Выражение вида *выражение op выражение*, где *op* — один из следующих бинарных операторов:
 - `+` `—` `*` `/`: арифметические операции над целыми числами;
 - `&` `|`: логические операции `And` и `Or` (побитовые над целыми числами);
 - `<` `>` `=`: операторы сравнения.
- Выражение в скобках: (*выражение*).

Приоритет операторов и порядок выполнения: приоритет операторов не определяется языком, за исключением того, что выражения в круглых скобках вычисляются первыми. Таким образом, значение выражения $2 + 3 * 4$ непредсказуемо, тогда как $2 + (3 * 4)$ гарантированно даст результат 14. Компилятор Jack, поставляемый в рамках программного обеспечения «От Nand до “Тетриса”» (а также компилятор,

который мы будем разрабатывать в главах 10–11), оценивает выражения слева направо, поэтому выражение $2 + 3 * 4$ даст результат 20. Опять же, если вы хотите получить алгебраически правильный результат, используйте выражение $2 + (3 * 4)$.

Необходимость использования круглых скобок для определения приоритетов операторов делает выражения на языке Jack немного громоздкими. Но мы намеренно решили отказаться от формального средства определения приоритетов операторов, поскольку это упрощает реализацию компиляторов Jack. При желании можно разработать свою версию компилятора Jack, определяющую приоритет операторов и добавляющую это средство в документацию языка.

Высказывание	Синтаксис	Описание
let	let varName = выражение; или let varName[выражение1] = выражение2;	Операция присваивания. Переменные могут быть статическими, локальными, полевыми или параметрами
if	if (выражение) { высказывание1; } else { высказывание2; }	Типичное if-высказывание. Фигурные скобки обязательны, даже если внутри него только одно высказывание
while	while (выражение) { высказывания; }	Типичное while-высказывание. Фигурные скобки обязательны, даже если внутри него только одно высказывание
do	do вызов-функции-или-метода;	Вызов функции или метода для их исполнения с игнорированием возвращаемого значения, если оно есть
return	return выражение; или return;	Возвращение значения из процедуры. Второй вариант используется для void-процедур. Конструкторы должны возвращать значение this

Иллюстрация 9.8. Высказывания в языке Jack.


```

...
do Foo.g()      ///Вызов функции g текущего класса
do Bar.h()      ///Вызов функции h класса Bar
do m()          ///Вызов метода m текущего класса
                к объекту this
do b.q()        ///Вызов метода q класса Bar
                к объекту b
let i = w(b.s(), Foo.t()) ///Вызов метода w
                        к объекту this,
                        ///Вызов метода s
                        класса Bar к объекту b,
                        ///Вызов функции или
                        конструктора t класса
                        Foo.
}
}

```

9.2.8. Создание и удаление объектов

Создание (конструкция) объекта происходит в два этапа. Сначала объявляется ссылочная переменная (указатель на объект). Чтобы завершить построение объекта (если это необходимо), программа должна вызвать конструктор из класса объекта. Таким образом, реализующий тип класс (например, `Fraction`) должен иметь хотя бы один конструктор. Конструкторы Jack могут иметь произвольные имена; по соглашению один из них называется `new`.

Объекты создаются и присваиваются переменным с помощью идиомы `let varName = className.constructorName (exp1, exp2, ..., expn)`, например, `let c = Circle.new (x, y, 50)`. Конструкторы обычно включают в себя код, инициализирующий поля нового объекта значениями аргументов, передаваемых вызывающей функцией.

Когда объект больше не нужен, его можно утилизировать (удалить), чтобы освободить занимаемую им память. Например, предположим, что объект, на который указывает `c`, больше не нужен. Объект можно удалить из памяти вызовом функции `OS Memory.deAlloc (c)`.

Поскольку в Jack нет сборки мусора, наилучший практический совет состоит в том, что каждый класс, представляющий объект, должен иметь метод `dispose()`, который правильно реализует это удаление. Примеры приведены на иллюстрациях 9.3 и 9.4. Чтобы избежать утечек памяти, программистам Jack рекомендуется избавляться от объектов, когда они больше не нужны.

9.3. Написание приложений на языке Jack

Jack — язык общего назначения, который может быть реализован на различных аппаратных платформах. В рамках практического курса «От Nand до “Тетриса”» мы разрабатываем компилятор Jack для платформы Hack, и поэтому вполне естественно обсуждать приложения Jack в контексте Hack.

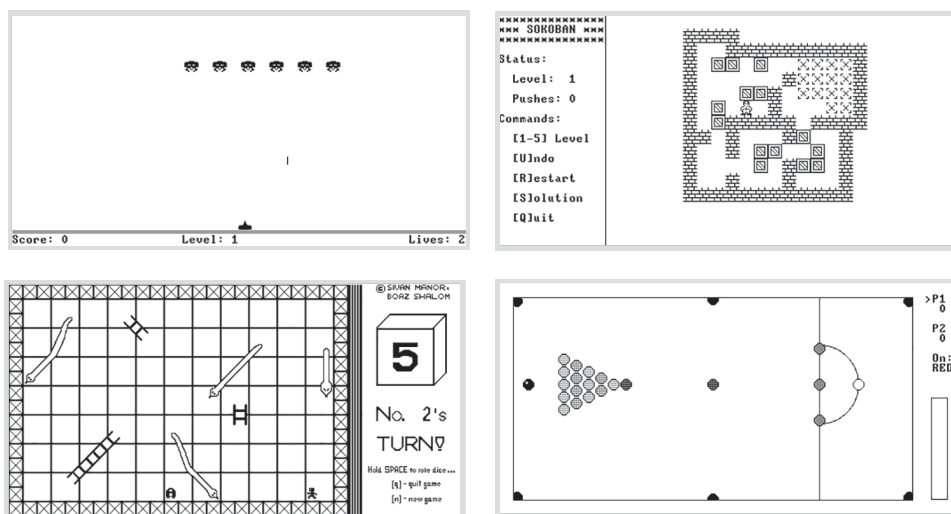


Иллюстрация 9.9. Скриншоты приложений, созданных на языке Jack и запущенных на компьютере Hack.

Примеры: на иллюстрации 9.9 показаны скриншоты четырех примеров программ Jack. Вообще говоря, платформа Jack/Hack хорошо подходит для создания простых интерактивных игр, таких как «Pong».

«Змейка», «Тетрис» и других подобных. Папка `projects/09/Square` на вашем компьютере содержит полный Jack-код простой интерактивной программы, позволяющей пользователю перемещать изображение квадрата на экране с помощью четырех клавиш со стрелками.

Неплохой способ научиться использовать Jack для написания интерактивных графических приложений — это просмотр и анализ исходного кода программы во время ее исполнения. Далее в главе мы опишем, как компилировать и выполнять программы Jack с помощью прилагаемых инструментов.

Разработка и реализация приложений: разработка программного обеспечения всегда должна основываться на тщательном планировании, особенно если она ведется на такой ограниченной аппаратной платформе, как компьютер Hack. Прежде всего разработчик программы должен учитывать физические ограничения аппаратного обеспечения и соответственным образом планировать свою работу. Так, например, размеры экрана компьютера ограничивают размер графических изображений, с которыми может работать программа. Точно так же следует принимать во внимание диапазон ввода/вывода языка и скорость выполнения программ. Иными словами, следует отталкиваться от реалистичного представления о том, что можно реализовать, а что нельзя.

Процесс разработки обычно начинается с концептуального описания поведения желаемой программы. В случае графических и интерактивных программ концепция может выражаться в виде сделанных от руки рисунков различных состояний экрана. Затем обычно продумывается объектно-ориентированная архитектура программы, подразумевающая определение *классов*, *полей* и *процедур*. Например, если программа должна позволять пользователю создавать квадратные объекты и перемещать их по экрану с помощью стрелок клавиатуры, то имеет смысл создать класс `Square`, реализующий эти операции с помощью таких методов, как `moveRight`, `moveLeft`, `moveUp` и `moveDown`, а также процедуру конструктора для создания квадратов и процедуру `dispose` для их утилизации. Кроме того, имеет смысл

создать класс `SquareGame`, который будет осуществлять взаимодействие с пользователем, и класс `Main`, начинающий работу. После тщательного определения API этих классов можно будет приступить к их реализации, компиляции и тестированию.

Компиляция и выполнение программ Jack: все файлы `.jack`, составляющие одну программу, должны находиться в одной папке. При применении компилятора Jack к папке с программой каждый исходный файл `.jack` транслируется в соответствующий файл `.vm`, сохраняемый в той же папке с программой.

Самый простой способ выполнить или отладить скомпилированную программу Jack — загрузить папку с программой в эмулятор VM. Эмулятор загрузит все функции VM во всех файлах `.vm` в папке одну за другой. В результате получится (возможно, довольно длинный) список функций VM, перечисленных в панели кода эмулятора VM под их полными именами `fileName.functionName`. После того как вы дадите эмулятору команду выполнить программу, эмулятор запустит функцию `Sys.init`, которая вызовет функцию `Main.main` вашей программы Jack.

В качестве альтернативы можно воспользоваться транслятором VM (подобным тому, который вы разрабатывали в проектах 7–8) для трансляции скомпилированного кода VM, а также восьми поставляемых файлов ОС `tools/OS/*.vm` в один файл `.asm`, написанный на машинном языке Hack. Затем этот ассемблерный код можно запустить на прилагаемом эмуляторе процессора. Или же можно воспользоваться ассемблером (подобным тому, который вы разрабатывали в проекте 6) для дальнейшего преобразования файла `.asm` в двоичный код `.hack`. Далее можно загрузить компьютерную микросхему Hack (подобную той, что была собрана в проектах 1–5) в симулятор аппаратуры или воспользоваться встроенной микросхемой `Computer`, загрузить двоичный код в микросхему ROM и выполнить этот код.

Операционная система: программы на языке Jack широко используют стандартную библиотеку классов языка, которую мы также называем операционной системой (ОС). В проекте 12 вы разработаете

библиотеку классов ОС на языке Jack (подобно тому, как ОС Unix написана на языке C) и скомпилируете ее с помощью компилятора Jack. В результате компиляции вы получите восемь файлов `.vm`, содержащих реализацию ОС. Если поместить эти восемь файлов `.vm` в папку с программой, скомпилированному коду ВМ станут доступны все функции ОС, поскольку они принадлежат одной и той же кодовой базе (в силу принадлежности к одной и той же папке).

Но пока что вам не обязательно задумываться о реализации ОС. Поставляемый эмулятор ВМ, представляющий собой программу на языке Java, имеет встроенную Java-реализацию ОС Jack. Когда загруженный в эмулятор ВМ код вызывает функцию ОС, скажем, `Math.sqrt`, происходит одно из двух. Если функция ОС найдена в загруженной базе кода, эмулятор ВМ выполняет ее, как и любую другую функцию ВМ. Если функция ОС не найдена в загруженной кодовой базе, эмулятор выполняет ее встроенную реализацию.

9.4. Проект

В отличие от других проектов в этой книге, данный проект не требует создания аппаратного или программного модуля. Здесь вы должны выбрать какое-то приложение и собрать его на языке Jack на платформе Hack.

Задача: «скрытая цель» этого проекта — знакомство с языком Jack для двух задач: написания компилятора Jack в проектах 10 и 11 и написания операционной системы Jack в проекте 12.

Контракт: воспользоваться идеей или придумать свою идею приложения в виде простой компьютерной игры или интерактивной программы. Затем спроектировать и создать приложение.

Ресурсы: вам понадобится компилятор Jack в папке `tools/JackCompiler` для перевода вашей программы в набор `.vm` файлов, а также эмулятор ВМ в папке `tools/VMEulator` для запуска и тестирования скомпилированного кода.

Компиляция и запуск программы Jack

0. Создайте папку для вашей программы, которую в дальнейшем будем называть *папкой программы*.
1. Напишите свою программу Jack — множество из одного или нескольких классов Jack, каждый из которых хранится в отдельном текстовом файле *ClassName.jack*. Поместите все эти файлы *.jack* в папку программы.
2. Скомпилируйте папку с программой, используя прилагаемый компилятор Jack. Компилятор транслирует все классы *.jack*, найденные в папке, в соответствующие файлы *.vm*. Если появится сообщение об ошибке компиляции, отладьте программу и перекомпилируйте ее — и так до тех пор, пока сообщения об ошибках не перестанут появляться.
3. На этом этапе папка программы должна содержать исходные файлы *.jack* и скомпилированные файлы *.vm*. Чтобы протестировать скомпилированную программу, загрузите папку с программой в прилагаемый эмулятор VM и запустите загруженный код. В случае возникновения ошибок во время выполнения или нежелательного поведения программы исправьте соответствующий файл и вернитесь к шагу 2.

Примеры программ: в папке `nand2tetris/project/09` на вашем компьютере содержится исходный код полной интерактивной программы Jack (*Square*), состоящей из трех классов. Папка также содержит исходный код программ Jack, обсуждаемых в этой главе.

Растровый редактор изображений: если вы разрабатываете программу, которой требуется высокоскоростная графика, лучше всего разработать спрайты для отрисовки ключевых графических элементов программы. Например, вывод приложения *Sokoban*, показанный на иллюстрации 9.9, состоит из нескольких повторяющихся спрайтов. Если вы хотите создать такие спрайты и записать их непосредственно в карту памяти экрана (минуя сервис класса *OS Screen*, который может быть слишком медленным), вам пригодится инструмент `projects/09/BitmapEditor`.

Веб-версия проекта 9 доступна на сайте www.nand2tetris.org.

9.5 Перспектива

Jack — *объектно-ориентированный* язык, то есть он поддерживает объекты и классы, но не наследование. В этом отношении он находится где-то между такими процедурными языками, как Pascal или C, и такими объектно-ориентированными языками, как Java или C++. Безусловно, Jack проще любого из этих мощных и профессиональных языков программирования. Однако его синтаксис и семантика в своей основе похожи на синтаксис и семантику современных языков.

Некоторые особенности языка Jack оставляют желать лучшего. Например, его система примитивных типов довольно, если можно так выразиться, примитивна. Более того, это слабо типизированный язык, то есть в нем не обязательно соблюдать строгое соответствие типов в присваиваниях и операциях. Кроме того, у вас могут возникнуть вопросы, почему синтаксис языка Jack включает такие неуклюжие ключевые слова, как `do` и `let`; почему каждая процедура должна заканчиваться высказыванием `return`; почему язык не поддерживает приоритет операций и т. д. — список вопросов не ограничен.

Все эти несколько утомляющие особенности были введены в Jack с одной целью: упростить разработку простых и минимальных компиляторов Jack, чем мы и займемся в последующих двух главах. Так, например, синтаксический разбор (парсинг) высказывания (на любом языке) значительно облегчается, если первый токен высказывания указывает на тип этого высказывания. Именно поэтому в Jack используется ключевое слово `let` для префиксации операторов присваивания. Таким образом, пусть простота языка Jack и может доставлять некоторые неудобства во время написания приложений на нем, но вы мысленно поблагодарите язык за такую простоту при написании компилятора языка Jack.

Большинство современных языков поставляются с набором стандартных классов; это же верно и для Jack. В своей совокупности эти классы можно рассматривать как переносимую, ориентированную на язык операционную систему. Однако в отличие от стандартных библиотек профессиональных языков, включающих множество классов,

ОС Jack предоставляет минимальный набор служб, которых тем не менее достаточно для разработки простых интерактивных приложений.

Очевидно, что было бы неплохо расширить ОС Jack так, чтобы обеспечить параллельность для поддержки многопоточности, файловую систему для постоянного хранения данных, сокет для обмена данными и т. д. Теоретически все эти службы могут быть добавлены в ОС, но, возможно, читатели захотят отточить свои навыки программирования в другом месте. В конце концов, мы не ожидаем, что Jack настолько плотно войдет в вашу жизнь после завершения практического курса «От Nand до “Тетриса”». Поэтому лучше всего воспринимать платформу Jack/Hack как некую данную среду и постараться извлечь из нее максимум пользы. Именно так поступают программисты, когда пишут программное обеспечение для встроенных устройств и специализированных процессоров, работающих в ограниченных условиях. Вместо того чтобы рассматривать накладываемые платформой хоста ограничения как проблему, профессионалы рассматривают их как возможность проявить свою изобретательность и находчивость. Именно это вам и предстоит сделать в проекте 9.

10. Компилятор I: синтаксический анализ

Ни словесное украшение нельзя найти, не выработав
и не представив себе отчетливо мыслей, ни мысль не может об-
рести блеск без светоча слов.

— Цицерон (106–43 до н. э.)

В предыдущей главе был описан Jack — простой, объектно-ориентированный язык программирования с синтаксисом, похожим на синтаксис языка Java. В этой главе мы начнем создавать компилятор для него. Компилятор — это программа, которая переводит программы с исходного языка на целевой. Процесс такого перевода, называемый *компиляцией*, концептуально основан на двух различных задачах. Во-первых, нужно понять синтаксис исходной программы и, исходя из него, раскрыть семантику программы. Например, при анализе кода можно понять, что программа стремится объявить массив или манипулировать объектом. Поняв семантику выражения на исходном языке, мы сможем выразить ее с помощью синтаксиса целевого языка. В этой главе описывается первая задача, обычно называемая *синтаксическим анализом*; следующая за ней задача, *генерация кода*, рассматривается в следующей главе.

Как можно определить, что компилятор способен «понимать» программы? Если код, сгенерированный компилятором, делает то, что должен делать, мы оптимистично можем предположить, что компилятор работает правильно. Однако в этой главе мы создаем только модуль синтаксического анализатора компилятора без возможности

генерации кода. Если мы хотим провести модульное тестирование синтаксического анализатора в изоляции, нам придется придумать способ показать, что он действительно понимает исходную программу. Мы предлагаем следующее решение: синтаксический анализатор выводит XML-файл с содержимым, разметка которого отражает синтаксическую структуру исходного кода. Проверив сгенерированный XML-файл, мы сможем убедиться, что анализатор правильно разбирает входные программы.

Разработка компилятора с нуля — процесс, в котором отражены несколько фундаментальных тем компьютерной науки. Он требует использования методов синтаксического анализа и трансляции, применения таких классических структур данных, как деревья и хеш-таблицы, и использования рекурсивных алгоритмов компиляции. Как следствие, написание компилятора — довольно сложная задача. Однако, разделив создание компилятора на два независимых проекта (в действительности четыре, если считать главы 7 и 8) и разрешив модульную разработку и тестирование каждой части по отдельности, мы превращаем этот процесс в управляемый и самодостаточный вид деятельности.

Но зачем вам вообще заниматься созданием компилятора? Помимо субъективных факторов, таких как удовольствие от успешно выполненной работы и осознание своей компетентности, практическое понимание внутренних механизмов компиляции поможет вам повысить свое мастерство как программиста. Кроме того, те же правила и грамматики, которые используются для описания языков программирования, применяются в таких областях, как компьютерная графика, коммуникации и сети, биоинформатика, машинное обучение, наука о данных и технология блокчейна. И, конечно же, умения анализировать имеющиеся тексты и синтезировать новые тексты на основе их семантики требует динамично развивающаяся область *обработки естественных языков* — практическая и теоретическая дисциплина, лежащая в основе разработки интеллектуальных чат-ботов, роботизированных персональных помощников, языковых переводчиков и многих приложений искусственного интеллекта. Несмотря на то что большинство программистов не разрабатывают компиляторы в рамках

своей обычной работы, многим из них приходится разбирать и обрабатывать тексты и наборы данных со сложными и разнообразными структурами. Эти задачи можно решать эффективно и элегантно с помощью алгоритмов и техник, описанных в этой главе.

Мы начинаем с раздела «Основы», в котором рассматривается минимальный набор понятий, необходимых для создания синтаксического анализатора: лексический анализ, контекстно-свободные грамматики, деревья синтаксического анализа (парсинга) и алгоритмы разбора с рекурсивным спуском. Там мы зададим основу для раздела «Спецификация», в котором описывается грамматика языка Jack и вывод, который должен генерировать анализатор Jack. В разделе «Реализация» предлагается программная архитектура для построения анализатора Jack, а также соответствующий API. В разделе «Проект», как обычно, приводятся пошаговые инструкции и тестовые программы для создания синтаксического анализатора. В следующей главе этот анализатор будет расширен до полноценного компилятора.

10.1. Основы

Компиляция состоит из двух основных этапов: *синтаксического анализа* и *генерации кода*. Этап синтаксического анализа обычно делится на два подэтапа: *токенизация* (разбиение на лексемы) — группировка входных символов в «атомы» языка, называемые *токенами* (лексемами), и *синтаксический анализ* (*парсинг*) — группировка токенов в структурированные высказывания, имеющие смысл.

Задачи токенизации и парсинга совершенно не зависят от целевого языка, на который мы хотим перевести исходный текст. Поскольку в этой главе мы не занимаемся генерацией кода, то решили, что синтаксический анализатор будет выводить разобранную структуру входной программы в виде XML-файла. Это решение имеет два преимущества. Во-первых, выходной файл можно легко просмотреть и понять, правильно ли синтаксический анализатор разбирает исходные программы. Во-вторых, требование выводить этот файл в явном виде заставляет нас написать синтаксический анализатор в архитектуре, которую

впоследствии можно преобразовать в полномасштабный компилятор. И действительно, как показано на иллюстрации 10.1, в следующей главе мы расширим разработанный в этой главе синтаксический анализатор до полномасштабного компилятора, способного генерировать исполняемый VM-код, а не пассивный XML-код.

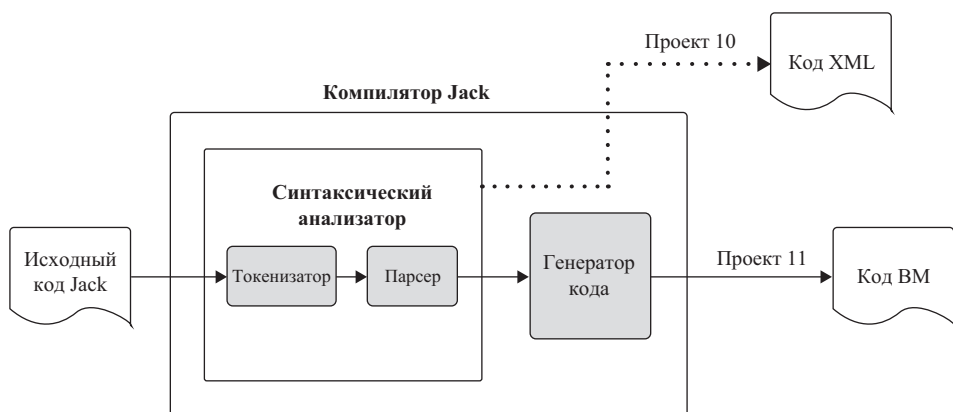


Иллюстрация 10.1. Поэтапная реализация плана по созданию компилятора Jack.

В этой главе мы сосредоточимся только на модуле синтаксического анализатора компилятора, задача которого — *понять структуру программы*. Эта концепция нуждается в некотором пояснении. Когда исходный код компьютерной программы читает человек, он может сразу соотнести его со структурой программы. У него это получается, потому что у него имеется мысленный образ *грамматики* языка. В частности, он ощущает, какие программные конструкции допустимы, а какие нет. Полагаясь на свое понимание грамматики, человек определяет, где в тексте начинаются и заканчиваются объявления классов и методов, где в нем содержатся высказывания и выражения, как они строятся и т. д. Для распознавания этих языковых конструкций, которые бывают и вложенными, люди рекурсивно сопоставляют их с рядом текстовых шаблонов, допускаемых грамматикой языка.

Синтаксические анализаторы можно разрабатывать аналогичным образом, в соответствии с заданной *грамматикой* — набором правил,

определяющих синтаксис языка программирования. Понять — то есть разобрать (парсировать) — заданную программу означает определить точное соответствие между текстом программы и правилами грамматики. Для этого нужно сначала преобразовать текст программы в список *токенов*, что мы сейчас и рассмотрим.

10.1.1. Лексический анализ

Спецификация каждого языка программирования включает типы *токенов*, или слов, которые распознает язык. В языке Jack токены делятся на пять категорий: *ключевые слова* (например, `class` и `while`), *символы* (например, `+` и `<`), *целочисленные константы* (например, `17` и `314`), *строковые константы* (например, `"FAQ"` или `"Часто задаваемые вопросы"`) и *идентификаторы* (текстовые метки, используемые для именования переменных, классов и процедур). В совокупности все токены, определяемые этими лексическими категориями, можно назвать *лексиконом* языка.

В самом простом виде компьютерная программа представляет собой поток (последовательность) символов, хранящихся в текстовом файле. Первым шагом в анализе синтаксиса программы будет группировка символов в токены согласно определениям лексикона языка при игнорировании пробелов и комментариев. Данная задача называется *лексическим анализом*, *сканированием* или *токенизацией* — все эти термины означают одно и то же.

После токенизации в качестве основных атомов программы рассматриваются уже ее токены, а не символы. Таким образом, на вход собственно компилятора подается поток токенов.

На иллюстрации 10.2 представлен лексикон языка Jack и показана токенизация типичного сегмента кода. Эта версия токенизатора выводит токены, а также их лексические классификации.

Токенизация — простая, но важная задача. Имея под рукой лексикон языка, несложно написать программу, превращающую любой заданный поток символов в поток токенов, что и будет первым шагом на пути к созданию синтаксического анализатора.

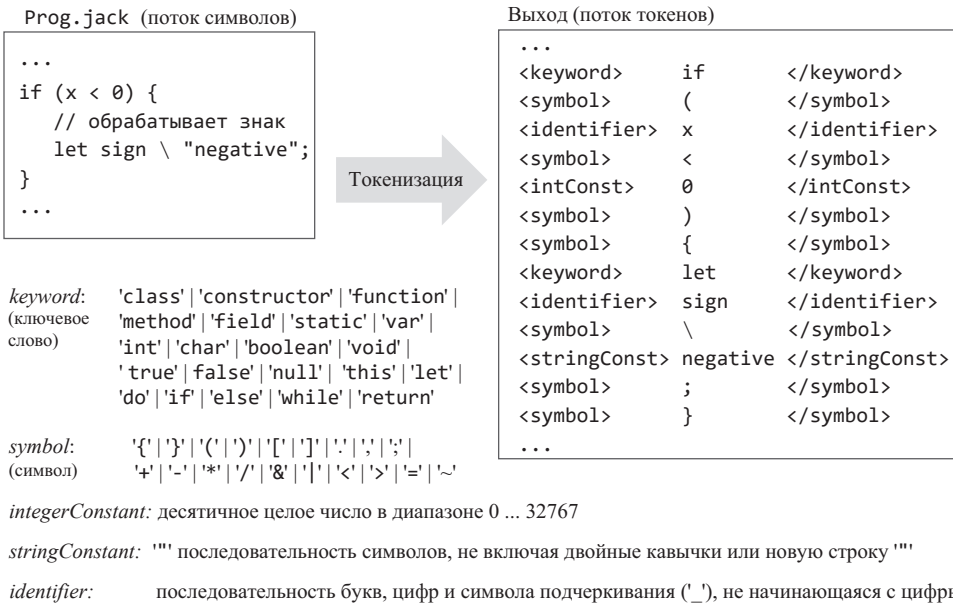


Иллюстрация 10.2. Определение лексикона языка Jack и пример лексического анализа входящего потока.

10.1.2. Грамматики

Как только мы научимся оценивать текст как поток токенов, или слов, то можем попытаться сгруппировать эти слова в грамматически правильные предложения. Например, когда мы слышим, что «Боб получил работу», мы одобрительно киваем, в то время как, услышав такие фразы, как «Боб работа получить» или «Работой Бобу получила», мы озадаченно смотрим на собеседника. Мы выполняем задачи синтаксического анализа, не задумываясь о них, поскольку мозг научился определять смысл последовательностей слов и их частей согласно шаблонам, допускаемым грамматикой нашего языка. Грамматики языков программирования намного проще грамматик естественных языков. Пример грамматики языка программирования показан на иллюстрации 10.3.

Грамматика Jack (подмножество)

<i>statements</i> :	<i>statement</i> *
<i>statement</i> :	<i>letStatement</i> <i>ifStatement</i> <i>whileStatement</i>
<i>letStatement</i> :	'let' <i>varName</i> '=' <i>expression</i> ';'
<i>ifStatement</i> :	'if' '(' <i>expression</i> ')' ' <i>{</i> ' <i>statements</i> ' <i>}</i> '
<i>whileStatement</i> :	'while' '(' <i>expression</i> ')' ' <i>{</i> ' <i>statements</i> ' <i>}</i> '
<i>expression</i> :	<i>term</i> (<i>op term</i>)?
(выражение)	
<i>term</i> :	<i>varName</i> <i>constant</i>
<i>varName</i> :	строка, не начинающаяся с цифры
<i>constant</i> :	неотрицательное целое число
(константа)	
<i>op</i> :	'+' '-' '=' '>' '<'

Примеры кода

let x \ 100;	✓
let x \ x + 1;	✓
if (x \ 1) let x \ 100; let x \ x + 1; }	✗
while (lim < 100) { if (x \ 1) { let z \ 100; while (z > 0) { let z \ z - 1; } } let lim \ lim + 10; }	✓

Иллюстрация 10.3. Подмножество грамматики языка Jack и сегменты кода на языке Jack, допустимые или недопустимые согласно грамматике.

Грамматика описывается на *метаязыке*, то есть на языке, описывающем язык. Теория компиляции изобилует различными определениями и терминами, призванными формализовать рассуждения о грамматиках, языках и метаязыках. Некоторые из формальных определений просто до жути формальны. В попытке упростить ситуацию мы в рамках практического курса «От Nand до “Тетриса”» рассматриваем грамматику как набор правил. Каждое правило состоит из двух частей: правой и левой. В левой части указывается название правила, не являющееся частью языка. Это название придумывает человек, описывающий грамматику, и поэтому его конкретная формулировка не так уж и важна. Если мы, например, заменим одно имя правила на другое во всей грамматике, то сама грамматика от этого не изменится (хотя, возможно, и станет менее читаемой).

Правая часть правила описывает задающий данное правило языковой шаблон. Этот шаблон состоит из трех следующих слева направо «строительных блоков»: *терминалов*, *нетерминалов* и *квалификаторов*.

Терминалы — это токены, нетерминалы — имена других правил, а квалификаторы представлены пятью символами `|`, `*`, `?`, `(` и `)`. Терминальные элементы, такие как `'if'`, выделяются жирным шрифтом и заключаются в одинарные кавычки; нетерминальные элементы, такие как *выражение*, выделяются курсивом; квалификаторы выделяются обычным шрифтом. Например, правило *ifStatement*: `'if' '(' expression ')' '{' statements'}` предусматривает, что каждый допустимый экземпляр *ifStatement* должен начинаться с токена `if`, за которым следует токен `(`, за которым следует допустимый экземпляр выражения *expression* (определенного в другой части грамматики), после чего следует токен `)`, затем токен `{`, затем допустимый экземпляр высказываний *statements* (определенных в другой части грамматики), после чего следует токен `}`.

Если шаблон можно разобрать более чем одним способом, мы используем классификатор `|` для перечисления альтернатив. Например, правило *высказывание*: `letStatement | ifStatement | whileStatement` предусматривает, что *высказывание* (*statement*) может быть либо *letStatement*, либо *ifStatement*, либо *whileStatement*.

Квалификатор `*` используется для обозначения «0, 1 или более раз». Например, правило *statements*: `statement *` предусматривает 0, 1 или более экземпляров элемента *statement* (*высказывание*). Аналогично классификатор `?` используется для обозначения «0 или 1 раз». Например, правило *statement*: `term (op term)?` говорит о том, что выражение — это *term*, за которым может следовать или не следовать последовательность *op term*. Отсюда следует, что *expression* (*выражением*) можно назвать как *x*, так и *x + 17*, *5 * 7* или *x < y*. Квалификаторы `(` и `)` используются для группировки элементов грамматики. Например, `(op term)` говорит о том, что в контексте данного правила *op*, за которым следует *term*, должен рассматриваться как один грамматический элемент.

10.1.3. Синтаксический разбор (парсинг)

Синтаксические грамматики по своей сути рекурсивны. Точно так же как правильным считается предложение «Боб получил работу, которую предложила Алиса», так считается правильным и высказывание `if (x < 0) { if (y > 0) { ... } }`. Как же определить, допускается ли

грамматикой выражение, которое мы имеем на входе? Выделив первый токен и поняв, что перед нами шаблон `if`, сосредоточимся на правиле `ifStatement`: `'if' '(' expression ')' '{statements}'`. Согласно правилу за токеном `if` должен следовать токен `(`, затем `expression`, за которым следует токен `)`. И действительно, этим требованиям удовлетворяет элемент `(x < 0)`. Далее согласно правилу мы предвидим появление токена `{`, за которым следует `statements`, завершающееся токеном `}`. Согласно другому правилу `statements` — это 0 или более экземпляров элемента `statement`, а `statement`, в свою очередь, — это либо `letStatement`, либо `ifStatement`, либо `whileStatement`. И действительно, внутри фигурных скобок мы видим `if (y > 0) {...}`, то есть `ifStatement`.

Как видно, грамматика языка программирования позволяет четко и недвусмысленно определить, представляют ли вводимые данные допустимые элементы кода или нет*. В качестве побочного эффекта такого разбора синтаксический анализатор создает точное соответствие между заданными входными данными с одной стороны и синтаксическими шаблонами, допускаемыми правилами грамматики, с другой. Это соответствие можно представить в виде структуры данных, называемой «деревом парсера» («деревом синтаксического анализа») или «деревом деривации» вроде показанного на иллюстрации 10.4а. Если можно построить такое дерево, синтаксический анализатор считает входные данные корректными; в противном случае он может сообщить, что входные данные содержат синтаксические ошибки.

* И здесь кроется важнейшее различие между языками программирования и естественными языками. На естественных языках мы можем сказать что-то вроде: «*Whoever saves one life, saves the world entire*» («Кто жизнь одну спасает, тот мир спасает целиком весь»), хотя согласно правилам английского языка ставить прилагательное после существительного некорректно («*world entire*» вместо «*entire world*»). Но в данном конкретном случае это звучит вполне приемлемо. В отличие от языков программирования, естественные языки позволяют некоторую поэтическую свободу — дают своего рода лицензию на нарушение или варьирование грамматических правил, при условии, что пишущий знает, что делает. Такая свобода самовыражения делает естественные языки бесконечно богатыми.

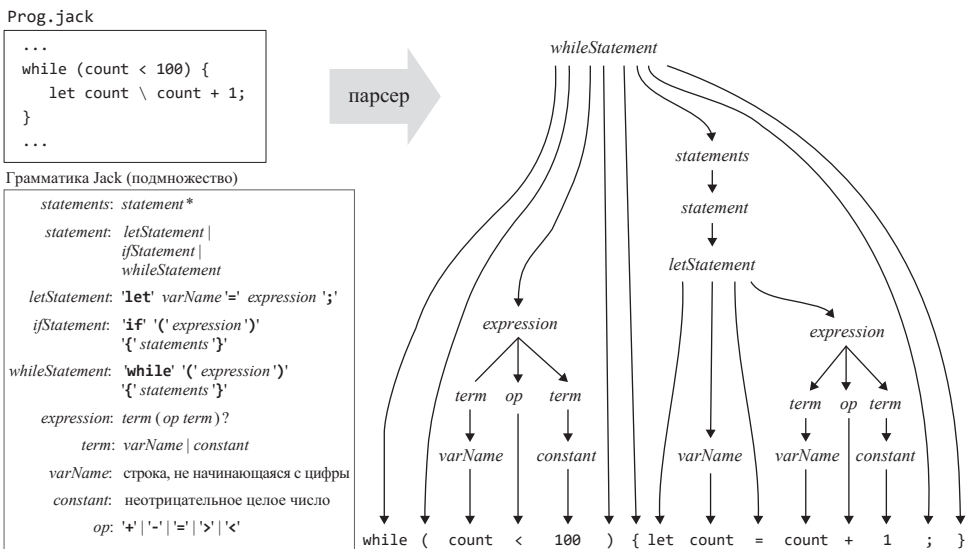


Иллюстрация 10.4а. Дерево парсера типичного сегмента кода. Процесс парсинга (синтаксического анализа) определяется правилами грамматики.

Как представить деревья парсера в текстовом виде? В рамках практического курса «От Nand до “Тетриса”» мы решили заставить синтаксический анализатор выводить XML-файл, формат которого отражает структуру дерева. Просматривая этот выходной XML-файл, мы можем убедиться, что синтаксический анализатор правильно разбирает входные данные. Пример разбора показан на иллюстрации 10.4б.

10.1.4. Парсер

Парсер (синтаксический анализатор) — это некий агент, работающий в соответствии с заданной грамматикой. Парсер принимает на вход поток токенов и пытается выдать на выходе дерево парсера, связанное с данным входным потоком. В нашем случае ожидается, что входные данные будут структурированы в соответствии с грамматикой языка Jack, а выходные данные будут записываться в формате XML. Заметим, однако, что методы разбора, которые мы сейчас опишем, применимы для работы с любым языком программирования и структурированным форматом файлов.

Prog.xml

```

...
<whileStatement>
  <keyword> while </keyword>
  <symbol> ( </symbol>
  <expression>
    <term> <varName> count </varName> </term>
    <op> <symbol> < </symbol> </op>
    <term> <constant> 100 </constant> </term>
  </expression>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <statements>
    <statement> <letStatement>
      <keyword> let </keyword>
      <varName> count </varName>
      <symbol> = </symbol>
      <expression>
        <term> <varName> count </varName> </term>
        <op> <symbol> + </symbol> </op>
        <term> <constant> 1 </constant> </term>
      </expression>
      <symbol> ; </symbol>
    </letStatement> </statement>
  </statements>
  <symbol> } </symbol>
</whileStatement>
...

```

Иллюстрация 10.46. Пример дерева парсера в XML.

Существует несколько алгоритмов построения деревьев синтаксического анализа. Метод «движения сверху вниз», также известный как «синтаксический разбор с рекурсивным спуском», пытается рекурсивно разобрать токенизированный вход с использованием вложенных структур, допустимых грамматикой языка. Реализовать такой алгоритм можно следующим образом. Для каждого нетривиального правила в грамматике мы в парсере записываем процедуру, предназначенную для разбора входных данных в соответствии с этим правилом. Например, грамматику, показанную на иллюстрации 10.3, можно реализовать с помощью набора процедур с именами `compileStatement`, `compileStatements`, `compileLet`, `compileIf`, ..., `compileExpression` и т. д. В названии процедур мы использовали глагол `compile` («компилировать»), а не `parse` («разбирать»), потому что в следующей главе расширим эту логику до полномасштабного механизма компиляции.

Логика разбора каждой процедуры `compilexxx` должна следовать синтаксическому шаблону, указанному в правой части правила `xxx`.

Возьмем для примера правило *whileStatement*: **'while'** '(' *expression* ')' '{' *statements* '}'. Согласно нашей схеме, это правило будет реализовано процедурой (подпрограммой) синтаксического анализа с именем *compileWhile*. Эта процедура должна реализовать логику деривации слева направо, заданную шаблоном **'while'** '(' *expression* ')' '{' *statements* '}'. Вот один из способов реализации этой логики с помощью псевдокода.

```
// Эта процедура реализует правило whileStatement:           //Вспомогательная процедура,
// 'while' '(' expression ')' '{ statements }'                //обрабатывающая
// Должна вызываться, если текущий токен = 'while' .         //текущий токен и переходящая
                                                             //к следующему токenu.

compileWhile:                                               process(str) :
print("<whileStatement>")
process("while")
process("(")
compileExpression()
process(" ")
process("{")
compileStatements()
process("}")
print("</whileStatement>")

if (currentToken == str)
printXMLToken(str)
else
print("syntax error")
//Получает следующий токен
currentToken =
tokenizer.advance()
```

Этот процесс разбора будет продолжаться до тех пор, пока не будут разобраны все части *expression* и *statements* оператора *while*. Конечно, часть *statements* может содержать в себе оператор *while* более низкого уровня, и в этом случае разбор будет продолжаться рекурсивно.

Только что приведенный пример показывает реализацию относительно простого правила, логика выведения (деривации) значения которого влечет за собой простой случай прямолинейного разбора. В целом же грамматические правила могут быть более сложными. Рассмотрим, например, следующее правило, определяющее статические переменные уровня класса и полевые переменные уровня экземпляра в языке Jack:

```
classVarDec:      ('static' | 'field') type
                   varName (' , ' varName)* ';' ;'
(где type и varName определены в других правилах
грамматики Jack).
```

Примеры: static int count;
 static char a, b, c;
 field boolean sign;
 field int up, down, left, right.

Это правило при разборе порождает две проблемы, выходящие за рамки прямолинейного разбора. Во-первых, допускает в качестве первого токена либо *static*, либо *field*. Во-вторых, разрешает несколько объявлений переменных. В качестве решения обеих проблем можно предложить следующее: реализация соответствующей процедуры *compileClassVarDec* может: 1) обрабатывать первый токен (*static* или *field*) напрямую, без вызова вспомогательной процедуры, и 2) использовать цикл для обработки всех объявлений переменных, которые содержатся во входных данных. Вообще говоря, разные правила грамматики влекут за собой немного разные реализации синтаксического анализа. В то же время все они следуют одному и тому же договору: каждая процедура *compilexxx* должна получить из входных данных и обрабатывать все составляющие *xxx* токены, продолжать работу токенизатора точно за этими токенами и на выходе давать дерево синтаксического анализа *xxx*.

Рекурсивные алгоритмы синтаксического анализа просты и элегантны. Если язык простой, то достаточно просмотреть один токен, чтобы понять, какое правило синтаксического анализа следует вызвать следующим. Например, если текущий токен — *let*, мы понимаем, что перед нами высказывание *letStatement*; если текущий токен — *while*, то перед нами высказывание *whileStatement*, и т. д. Действительно, в простой грамматике, показанной на иллюстрации 10.3, достаточно заглянуть вперед всего на один лишь токен, чтобы четко и однозначно решить, какое правило использовать следующим. Грамматики, обладающие этим лингвистическим свойством, называются грамматиками *LL* (1). Их можно просто и элегантно обрабатывать алгоритмами рекурсивного спуска без возвращения.

Термин *LL* происходит от того наблюдения, что грамматика разбирает входные данные *слева направо*, выполняя деривацию входных данных по *самоу левому* токenu. Параметр (1) сообщает, что достаточно

заглянуть на 1 токен вперед, чтобы узнать, какое правило синтаксического анализа следует вызвать следующим. Если этого токена недостаточно, чтобы решить, какое правило использовать, можно заглянуть еще на один токен вперед. Если такой шаг устраняет неоднозначность, то считается, что это синтаксический анализатор *LL* (2). Если нет, можно заглянуть еще на один токен вперед, и т. д. Очевидно, что по мере того, как приходится заглядывать вперед все дальше и дальше по потоку токенов, ситуация усложняется и требует более сложного синтаксического анализатора.

Полная грамматика языка Jack, которую мы сейчас представим, это как раз грамматика *LL* (1), за исключением одного случая, с которым можно легко справиться. Таким образом, язык Jack хорошо поддается разбору с помощью синтаксического анализатора (парсера) с рекурсивным спуском, который и лежит в основе проекта 10.

10.2. Спецификация

Этот раздел состоит из двух частей. Сначала мы приводим спецификацию грамматики языка Jack. Затем разбираем спецификацию определения синтаксического анализатора, предназначенного для разбора программ в соответствии с этой грамматикой.

10.2.1. Грамматика языка Jack

Функциональная спецификация языка Jack, представленная в главе 9, предназначалась для программистов, составляющих программы на данном языке; теперь мы приводим формальную спецификацию языка Jack, предназначенную для разработчиков компиляторов этого языка. В спецификации языка, или его *грамматике*, используется следующая нотация:

'xxx'	:	языковые токены, передаваемые буквально;
xxx	:	имена терминалов и нетерминалов;
()	:	используется для группировки;
$x \mid y$	:	либо x , либо y ;

$x\ y$: за x следует y ;
 $x?$: x появляется 0 или 1 раз;
 x^* : x появляется 0 или более раз.

Лексические элементы: в языке Jack имеются пять типов терминальных элементов (токенов):	
<i>keyword:</i>	'class' 'constructor' 'function' 'method' 'field' 'static' 'var' 'int' 'char' 'boolean' 'void' 'true' 'false' 'null' 'this' 'let' 'do' 'if' 'else' 'while' 'return'
<i>symbol:</i>	{' ' '(' ')' '[' ']' '.' ',' ';' '+' '-' '*' '/' '%' ' ' '<' '>' '=' '~' }
<i>integerConstant:</i>	десятичное число в диапазоне 0... 32767
<i>StringConstant:</i>	"" последовательность символов, не включая двойные кавычки или новую строку ""
<i>identifier:</i>	последовательность букв, цифр и символа подчеркивания ('_'), не начинающаяся с цифры
Структура программы: программа на языке Jack — это набор классов, каждый из которых находится в отдельном файле. Единицей компиляции является класс. Класс — это последовательность токенов, как показано ниже:	
<i>class:</i>	'class' className '{ classVarDec* subroutineDec* '}'
<i>classVarDec:</i>	('static' 'field') type varName (',' varName)* ';'
<i>type:</i>	'int' 'char' 'boolean' className
<i>subroutineDec:</i>	('constructor' 'function' 'method') ('void' type) subroutineName ('(' parameterList ')') subroutineBody
<i>parameterList:</i>	((type varName) (',' type varName)*)?
<i>subroutineBody:</i>	'{' varDec* statements '}'
<i>varDec:</i>	'var' type varName (',' varName)* ';'
<i>className:</i>	identifier
<i>subroutineName:</i>	identifier
<i>varName:</i>	identifier
Высказывания (statements):	
<i>statements:</i>	statement*
<i>statement:</i>	letStatement ifStatement whileStatement doStatement returnStatement
<i>letStatement:</i>	'let' varName ('[' expression ']')? '=' expression ';'
<i>ifStatement:</i>	'if' '(' expression ')' '{ statements '}' ('else' '{ statements '}')?
<i>whileStatement:</i>	'while' '(' expression ')' '{ statements '}'
<i>doStatement:</i>	'do' subroutineCall ';'
<i>returnStatement:</i>	'return' expression? ';'
Выражения (expressions):	
<i>expression:</i>	term (op term)*
<i>term:</i>	integerConstant stringConstant keywordConstant varName varName '[' expression ']' '(' expression ')' (unaryOp term) subroutineCall
<i>subroutineCall:</i>	subroutineName '(' expressionList ')' (className varName) '.' subroutineName '(' expressionList ')'
<i>expressionList:</i>	(expression (',' expression)*)?
<i>op:</i>	'+' '-' '*' '/' '%' ' ' '<' '>' '='
<i>unaryOp:</i>	'-' '~'
<i>keywordConstant:</i>	'true' 'false' 'null' 'this'

Иллюстрация 10.5. Грамматика языка Jack.

Полная грамматика языка Jack с учетом этой нотации представлена на иллюстрации 10.5.

10.2.2. Синтаксический анализатор для языка Jack

Синтаксический анализатор — это программа, которая выполняет как токенизацию, так и синтаксический разбор (парсинг). В практическом курсе «От Nand до “Тетриса”» основная цель синтаксического анализатора — обработать программу на языке Jack и понять ее синтаксическую структуру в соответствии с грамматикой языка Jack. Под *пониманием* мы подразумеваем, что синтаксический анализатор в каждый момент разбора (парсинга) должен определять структурную идентичность обрабатываемого элемента, то есть идентифицировать его как выражение, оператор, имя переменной и т. д. Синтаксический анализатор должен обладать этим синтаксическим знанием в полном рекурсивном смысле. Без этого невозможно перейти к генерации кода — конечной цели процесса компиляции.

Использование: синтаксический анализатор принимает один аргумент командной строки, как показано ниже:

```
prompt> JackAnalyzer source,
```

где *source* — это либо имя файла вида *Xxx.jack* (расширение обязательно), либо имя папки (в этом случае расширение отсутствует), содержащей один или несколько файлов *.jack*. Имя файла/папки может содержать путь к файлу. Если путь не указан, анализатор работает с текущей папкой. Для каждого файла *Xxx.jack* анализатор создает выходной файл *Xxx.xml* и записывает в него разобранные элементы. Выходной файл создается в той же папке, что и входной. Если в папке уже есть файл с таким именем, он будет перезаписан.

Ввод: файл *Xxx.jack* представляет собой поток символов. Если это допустимая с точки зрения грамматика программа, файл можно

преобразовать в поток допустимых токенов согласно лексической спецификации Jack. Токены могут разделяться произвольным количеством пробелов, символов новой строки и комментариев, которые игнорируются. Существуют три возможных формата комментариев: `/*` комментарий до закрытия `*/`, `/**` API-комментарий до закрытия `*/` и `//` комментарий до конца строки.

Данные на выходе: синтаксический анализатор выдает XML-описание входного файла следующим образом. Для каждого терминального элемента (токена) типа `xxx`, встречающегося во входном файле, синтаксический анализатор создает («печатает») высказывание типа `<xxx> token </xxx>`, где `xxx` — один из тегов `keyword`, `symbol`, `integerConstant`, `stringConstant` или `identifier`, представляющий один из пяти типов распознаваемых языком Jack токенов. При обнаружении нетерминального языкового элемента `xxx` синтаксический анализатор обрабатывает его с помощью следующего псевдокода:

```
print("<xxx>"),
```

Рекурсивный код обработки тела элемента `xxx`:

```
print("</xxx>"),
```

где `xxx` — один из следующих (и только следующих) тегов: `class`, `classVarDec`, `subroutineDec`, `parameterList`, `subroutineBody`, `varDec`, `statements`, `letStatement`, `ifStatement`, `whileStatement`, `doStatement`, `returnStatement`, `expression`, `term`, `expressionList`.

Для упрощения при выводе XML не учитываются в явном виде следующие правила грамматики Jack: *type*, *className*, *subroutineName*, *varName*, *statement*, *subroutineCall*. Мы объясним это в следующем разделе, когда будем обсуждать архитектуру нашего механизма компиляции.

10.3. Реализация

В предыдущем разделе были определены действия синтаксического анализатора, но при этом мало было сказано о его реализации. В этом разделе описывается построение такого анализатора. Предлагаемая нами реализация основана на трех модулях.

- `JackAnalyzer`: основная программа, настраивающая и вызывающая другие модули.
- `JackTokenizer`: токенизатор.
- `CompilationEngine`: рекурсивный нисходящий синтаксический анализатор.

В следующей главе мы расширим эту программную архитектуру двумя дополнительными модулями, обрабатывающими семантику языка: *таблицей символов* и *программой записи кода ВМ*. На этом будет завершено создание полноценного компилятора для языка Jack.

Поскольку основной модуль, управляющий процессом синтаксического анализа в этом проекте, в конечном счете будет управлять и общим процессом компиляции, мы назвали его `CompilationEngine`.

JackTokenizer

Этот модуль игнорирует все комментарии и пробелы во входном потоке и обеспечивает доступ к входным данным по одному токenu за раз. Кроме того, он анализирует и предоставляет тип каждого токена согласно определению грамматики Jack.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Входной файл / поток	—	Открывает входной файл / поток .jack и готовится к его токенизации
hasMoreTokens	—	boolean	Есть ли еще токены на входе?
advance	—	—	Получает следующий токен из вво- да и делает его текущим токеном. Этот метод должен вызываться только в том случае, если значение hasMoreTokens равно true. Изначально текущего токена нет
tokenType	—	KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST	Возвращает тип текущего токена в виде константы
keyword	—	CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS	Возвращает ключевое слово, ко- торое является текущим токеном, в виде константы. Этот метод должен вызываться только в том случае, если tokenType — это KEYWORD
symbol	—	char	Возвращает символ, который яв- ляется текущим токеном. Вызы- вается только в том случае, если tokenType — это SYMBOL
identifier	—	string	Возвращает строку, которая яв- ляется текущим токеном. Вызы- вается только в том случае, если tokenType — это IDENTIFIER
intVal	—	int	Возвращает целочисленное зна- чение текущего токена. Вызы- вается только в том случае, если tokenType — это INT_CONST
stringVal	—	string	Возвращает строковое значение теку- щего токена, без открывающих и за- крывающих двойных кавычек. Вы- зывается только в том случае, если tokenType — это STRING_CONST

CompilationEngine

`CompilationEngine` — это основной модуль как описанного в этой главе синтаксического анализатора, так и описываемого в следующей главе полномасштабного компилятора. В синтаксическом анализаторе механизм компиляции выдает структурированное представление входного исходного кода, оформленного тегами XML. В компиляторе же механизм компиляции вместо этого выдает исполняемый VM-код. В обеих версиях представленные ниже логика разбора и API абсолютно одинаковы.

Модуль компиляции получает входные данные от `JackTokenizer` и направляет их в выходной файл. Выходные данные генерируются серией процедур `compilexxx`, каждая из которых предназначена для компиляции определенной конструкции `xxx` языка Jack. Каждая процедура `compilexxx` должна получить данные со входа и обработать все составляющие `xxx` токены, перейти к следующему токenu и вывести синтаксический анализ `xxx`. Как правило, каждая процедура `compilexxx` вызывается только в том случае, если текущий токен — `xxx`.

Правила грамматики, не имеющие соответствующих процедур `compilexxx`: *type*, *className*, *subroutineName*, *varName*, *statement*, *subroutineCall*. Мы ввели эти правила, чтобы сделать грамматику Jack более структурированной. Как оказалось, логика разбора этих правил лучше обрабатывается процедурами, реализующими правила, которые на них ссылаются. Например, вместо того чтобы писать подпрограмму `compileType` всякий раз, когда в каком-то правиле `xxx` упоминается некий тип `type`, разбор возможных типов должен выполняться непосредственно процедурой `compilexxx`.

Прогнозирующий просмотр токенов: Jack — это почти *LL(1)*-язык: текущего токена достаточно для определения того, какую процедуру `CompilationEngine` вызывать следующей. Единственное исключение — парсинг элемента *term* (*термин*), который происходит только при разборе выражения *expression*. Для примера рассмотрим специально придуманное, но корректное выражение

`y + arr[5] - p.get(row) * count() - Math.sqrt(dist) / 2`. Это выражение состоит из шести элементов *term*: переменная `y`, элемент массива `arr[5]`, вызов метода `p.get(row)` для объекта `p`, вызов метода `count()` для объекта `this`, вызов функции (статический метод) `Math.sqrt(dist)` и константы `2`.

Предположим, что мы осуществляем синтаксический анализ этого выражения и текущий токен — один из идентификаторов `y`, `arr`, `p`, `count` или `Math`. В каждом из этих случаев мы знаем, что у нас есть *term*, начинающийся с идентификатора *identfier*, но мы не знаем, какой вариант синтаксического анализа использовать дальше. Само по себе это неприятно, но хорошая новость заключается в том, что для решения проблемы достаточно посмотреть на следующий токен.

Необходимость в такой нерегулярной операции прогнозирующего, или предварительного, просмотра возникает в `CompilationEngine` дважды: при разборе элемента *term*, что бывает только при разборе выражения *expression*, и при разборе элемента *subroutineCall*. При анализе грамматики Jack становится ясно, что *subroutineCall* появляется только либо в высказывании `do subroutineCall`, либо в элементе *term*.

Учитывая это, мы предлагаем разбирать высказывания `do subroutineCall` так, как если бы их синтаксис был `do expression`. Эта прагматичная рекомендация избавляет от необходимости дважды писать код для нерегулярного предварительного просмотра. Она также подразумевает, что парсинг *subroutineCall* теперь можно осуществлять непосредственно процедурой `compileTerm`. Короче говоря, мы локализовали необходимость составлять специальный код для нерегулярного предварительного просмотра и свели его к одной процедуре `compileTerm`, избавившись от необходимости прописывать процедуру `compileSubroutineCall`.

Процедура `compileExpressionList`: возвращает количество выражений в списке. Возвращаемое значение необходимо для генерации кода ВМ, как мы увидим, когда завершим разработку компилятора в проекте 11. В этом проекте мы не генерируем код ВМ, поэтому возвращаемое значение не используется и процедуры, вызывающие `compileExpressionList`, могут его игнорировать.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Входной файл / поток Выходной файл / поток	—	Создает новый механизм компиляции с заданными входными и выходными данными. Следующей вызываемой (модулем JackAnalyzer) процедурой должна быть compileClass
compileClass	—	—	Компилирует полностью класс
compileClassVarDec	—	—	Компилирует объявление статической или полевой переменной
compileSubroutine	—	—	Компилирует полностью метод, функцию или конструктор
compileParameterList	—	—	Компилирует (возможно, пустой) список параметров. Не обрабатывает токены включающих скобок (и)
compileSubroutineBody	—	—	Компилирует тело процедуры
compileVarDec	—	—	Компилирует объявление var
compileStatements	—	—	Компилирует последовательность высказываний. Не обрабатывает токены включающих фигурных скобок { и }
compileLet	—	—	Компилирует высказывание let
compileIf	—	—	Компилирует высказывание if
compileWhile	—	—	Компилирует высказывание while
compileDo	—	—	Компилирует высказывание do
compileReturn	—	—	Компилирует высказывание return
compileExpression	—	—	Компилирует выражение
compileTerm	—	—	Компилирует term. Если текущий токен — identifier, то процедура должна преобразовывать его в переменную, элемент массива или вызов процедуры. Для различения этих вариантов достаточно одного маркера, например, [, (или . Любой другой маркер не является частью этого term и не подразумевает перехода к следующему токenu

<code>compileExpressionList</code>	—	<code>int</code>	Компилирует (возможно, пустой) список выражений, разделенных запятой. Возвращает количество выражений в списке
------------------------------------	---	------------------	--

JackAnalyzer

Это основная программа, управляющая общим процессом синтаксического анализа с использованием сервисов `JackTokenizer` и `CompilationEngine`. Для каждого исходного файла `Xxx.jack` анализатор:

- 1) создает `JackTokenizer` из входного файла `Xxx.jack`;
- 2) создает выходной файл `Xxx.xml`;
- 3) использует `JackTokenizer` и `CompilationEngine` для разбора входного файла и записи разобранного кода в выходной файл.

Мы не предоставляем API для этого модуля, предлагая вам реализовать его по своему усмотрению. Помните, что первой вызываемой процедурой при компиляции файла `.jack` должна быть `compileClass`.

10.4. Проект

Задача: создать синтаксический анализатор, который разбирает программы на языке Jack в соответствии с грамматикой языка Jack. Вывод анализатора должен представлять собой запись формата XML, как указано в разделе 10.2.2.

Эта версия синтаксического анализатора предполагает, что исходный код Jack не содержит ошибок. Проверку на наличие ошибок и обработку ошибок можно будет добавить в последующие версии анализатора, но в проект 10 они не входят.

Ресурсы: основной инструмент в этом проекте — язык программирования, которым вы воспользуетесь для реализации

синтаксического анализатора. Также вам понадобится прилагаемая утилита `TextComparer`. Эта программа позволяет сравнивать файлы без учета пробелов. Она поможет сравнить выходные файлы, созданные вашим анализатором, с прилагаемыми файлами. Возможно, вам захочется также просмотреть эти файлы с помощью средств просмотра XML, а для этого подойдет любой стандартный веб-браузер.

Контракт: написать синтаксический анализатор для языка Jack и протестировать с помощью предоставленных тестовых файлов. Созданные вашим анализатором XML-файлы должны быть идентичны предоставленным тестовым файлам без учета пробелов.

Тестовые файлы: для тестирования мы предоставляем несколько файлов `.jack`. Программа `projects/10/Square` — это приложение из трех классов, позволяющее перемещать черный квадрат по экрану с помощью клавиш со стрелками на клавиатуре. Программа `projects/10/ArrayTest` — это приложение из одного класса, вычисляющее среднее значение заданной пользователем последовательности целых чисел с помощью обработки массивов. Обе программы обсуждались в главе 9, поэтому они должны быть вам знакомы. При этом обратите внимание на то, что мы внесли в исходный код некоторые безобидные изменения, чтобы синтаксический анализатор можно было полностью протестировать для всех аспектов языка Jack. Например, в файл `projects/10/Square/Main.jack` мы добавили статическую переменную, а также функцию с именем `more`, которые никогда не используются и не вызываются. Эти изменения позволят проверить, как анализатор обрабатывает элементы языка, не встречающиеся в оригинальных файлах `Square` и `ArrayTest`, такие как статические переменные, `else` и унарные операторы.

План разработки: мы предлагаем разрабатывать и тестировать анализатор в четыре этапа.

— Сначала напишите и протестируйте токенизатор Jack.

- Затем напишите и протестируйте базовый механизм компиляции, обрабатывающий все возможности языка Jack, за исключением выражений и высказываний (команд), ориентированных на массивы.
- Затем расширьте механизм компиляции для обработки выражений.
- Наконец, расширьте механизм компиляции для обработки высказываний, ориентированных на массивы.

Мы предоставляем файлы `.jack` для входных данных и файлы `.xml` для сравнения во время модульного тестирования каждого из четырех описанных ниже этапов.

10.4.1. Токенизатор

Реализуйте модуль `JackTokenizer`, описанный в разделе 10.3. Проверьте свою реализацию, написав базовую версию `JackAnalyzer` согласно следующему определению. Основная программа анализатора вызывается с помощью команды `JackAnalyzer source`, где `source` — это либо имя файла вида `Xxx.jack` (расширение обязательно), либо имя папки (в этом случае расширение отсутствует). В последнем случае папка содержит один или несколько файлов `.jack` и, возможно, другие файлы. Имя файла/папки может включать путь к файлу. Если путь не указан, анализатор работает с текущей папкой.

Анализатор обрабатывает каждый файл отдельно. В частности, для каждого файла `Xxx.jack` анализатор конструирует `JackTokenizer` для обработки входных данных и выходной файл для записи выходных данных. В этой первой версии анализатора выходной файл называется `XxxT.xml` (где `T` обозначает *токенизированный вывод*). Затем анализатор входит в цикл для перебора и обработки всех токенов во входном файле, по одному токenu за раз, с использованием методов `JackTokenizer`. Каждый токен должен выводиться отдельной строкой вида `<tokenType> token </tokenType>`, где `tokenType` — один из пяти возможных XML-тегов, кодирующих тип токена. Вот пример.

Ввод (например, Prog.jack)

```
//Комментарии и пробелы
// игнорируются
...
if (x < 0) {
    let quit = "yes";
}
...
```

Вывод JackAnalyzer (например, ProgT.xml)

```
<tokens>
...
<keyword> if </keyword>
<symbol> (</symbol>
<identifier> x </identifier>
<symbol> &lt; </symbol>
<integerConstant> 0 </
integerConstant>
<symbol>) </symbol>
<symbol> {</symbol>
<keyword> let </keyword>
<identifier> quit </identifier>
<symbol> = </symbol>
<stringConstant> yes </
stringConstant>
<symbol> ; </symbol>
<symbol> } </symbol>
...
</tokens>
```

Обратите внимание, что в случае *строковых констант* (*stringConstant*) программа игнорирует двойные кавычки. Это требование продиктовано дизайном.

Сгенерированный вывод имеет две тривиальные технические особенности, продиктованные соглашениями XML. Во-первых, XML-файл должен иметь некоторый начальный и конечный теги; этому условию удовлетворяют теги `<tokens>` и `</tokens>`. Во-вторых, четыре символа, используемые в языке Jack (`<`, `>`, `"`, `&`), также используются для XML-разметки; следовательно, они не могут появляться в XML-файлах в качестве данных. По соглашению анализатор представляет эти символы как `<`, `>`, `"` и `&` соответственно. Например, встречая символ `<` во входном файле, анализатор выводит строку `<symbol> < </symbol>`. Эта так называемая *управляющая последовательность* отображается программами просмотра XML в виде `<symbol> < </symbol>`, что нам и нужно.

Рекомендации по тестированию

- Для начала примените свой `JackAnalyzer` к одному из поставляемых файлов `.jack` и убедитесь, что он правильно работает с одним входным файлом.
- Затем примените свой `JackAnalyzer` к папке `Square`, содержащей файлы `Main.jack`, `Square.jack` и `SquareGame.jack`, а также к папке `TestArray`, содержащей файл `Main.jack`.
- Используйте прилагаемую утилиту `TextComparer` для сравнения выходных файлов, сгенерированных вашим `JackAnalyzer`, с поставляемыми файлами сравнения `.xml`. Например, сравните сгенерированный файл `SquareT.xml` с прилагаемым файлом сравнения `SquareT.xml`.
- Поскольку сгенерированные файлы и файлы сравнения имеют одинаковые имена, рекомендуется поместить их в отдельные папки.

10.4.2. Механизм компиляции `CompilationEngine`

Следующая версия вашего синтаксического анализатора должна уметь разбирать каждый элемент языка `Jack`, за исключением выражений и команд, ориентированных на массивы. Для этого реализуйте описанный в разделе 10.3 модуль `CompilationEngine`, за исключением процедур, которые работают с выражениями и массивами. Проверьте реализацию следующим образом.

Для каждого файла `Xxx.jack` анализатор конструирует `JackTokenizer` для обработки входных данных и выходной файл для записи выходных данных с именем `Xxx.xml`. Затем анализатор вызывает процедуру `compileClass` модуля `CompilationEngine`. С этого момента процедуры `CompilationEngine` должны рекурсивно вызывать друг друга, выдавая текст в виде XML-разметки, подобный тому, что показан на иллюстрации 10.46.

Протестируйте эту версию `JackAnalyzer`, применив ее к папке `ExpressionlessSquare`. Данная папка содержит версии файлов `Square.jack`, `SquareGame.jack` и `Main.jack`, в которых

каждое выражение в исходном коде было заменено одним идентификатором (именем переменной в области видимости). Например:

Папка Square:

```
// Square.jack
...
method void incSize() {
if ((y + size) < 254) & ((x + size) < 510)
do erase();
let size = size + 2;
do draw();
}
return;
}
...
```

Папка ExpressionlessSquare:

```
//Square.jack
...
method void incSize() {
if (x) {
do erase();
let size = size
do draw();
}
return
}
...
```

Обратите внимание, что замена выражений переменными приводит к бессмысленному коду. Это нормально, поскольку для проекта 10 семантика программы не имеет значения. Бессмысленный код синтаксически корректен, и это все, что важно для тестирования синтаксического анализатора. Обратите также внимание, что исходные файлы и файлы без выражений имеют одинаковые имена, но расположены в разных папках. Используйте прилагаемую утилиту `TextComparer` для сравнения выходных файлов, созданных вашим `JackAnalyzer`, с прилагаемыми файлами сравнения `.xml`.

Затем завершите разработку процедур `CompilationEngine`, которые работают с выражениями, и протестируйте их, применив свой `JackAnalyzer` к папке `Square`. Наконец, завершите разработку процедур, которые работают с массивами, и протестируйте их, применив `JackAnalyzer` к папке `ArrayTest`.

Веб-версия проекта 10 доступна на сайте www.nand2tetris.org.

10.5. Перспектива

Хотя структуру компьютерных программ удобно описывать с помощью деревьев парсера и XML-файлов, важно понимать, что компиляторы не обязательно должны явно поддерживать такие структуры данных. Например, описанный в этой главе алгоритм парсинга разбирает входные данные по мере их чтения и не хранит в памяти всю входную программу. В целом существуют два типа стратегий выполнения такого разбора. Более простая стратегия работает «сверху вниз», и именно она представлена в этой главе. Более продвинутые алгоритмы синтаксического анализа, работающие «снизу вверх», здесь описаны не были, поскольку они требуют более подробной разработки теории компиляции.

И действительно, в этой главе мы обошли стороной формальную теорию языка, обычно изучаемую на курсах компиляции. Кроме того, выбрали простой синтаксис для языка Jack — такой, который легко компилируется с помощью методов рекурсивного спуска. Так грамматика языка Jack не подразумевает методов определения старшинства операторов при оценке алгебраических выражений, например, того, что операция умножения выполняется перед операцией сложения. Это позволило нам избежать обращения к более мощным, но и более сложным алгоритмам синтаксического анализа по сравнению с представленными в этой главе элегантными методами нисходящего синтаксического анализа.

Каждому программисту приходится сталкиваться с безобразной системой обработки ошибок компиляции, характерной для многих компиляторов. Как выясняется, диагностика и исправление ошибок — весьма сложная проблема. Во многих случаях влияние ошибки обнаруживается через несколько или множество строк кода после того, как она была допущена. Поэтому сообщения об ошибках иногда бывают довольно непонятными и недружелюбными. И действительно, одна из характеристик компиляторов, сильно отличающаяся в разных программах, — это их способность диагностировать ошибки и помогать их исправлять. Для этого компиляторы сохраняют части

дерева парсера в памяти и дополняют дерево аннотациями, помогающими определить источник ошибок и при необходимости отследить процесс диагностики. В практическом курсе «От Nand до “Тетриса”» мы обходимся без подобных расширений, исходя из того предположения, что исходные файлы, обрабатываемые компилятором, не содержат ошибок.

Еще одна тема, которую мы почти не упомянули, — это изучение синтаксиса и семантики языков программирования в сферах информатики и когнитивистики. Свойствами классов языков, а также метаязыками и формальными средствами их описания занимается широко развитая теория формальных и естественных языков. Это одна из тех областей, где компьютерная наука встречается с изучением человеческих языков, что открывает очень интересные направления исследований и практики, таких как компьютерная лингвистика и обработка естественного языка.

Наконец, стоит упомянуть, что синтаксические анализаторы обычно не бывают самостоятельными программами и редко пишутся с нуля. Вместо этого программисты обычно создают токенизаторы и синтаксические анализаторы с помощью различных инструментов *генерации компиляторов*, таких как LEX (от *LEXical analysis*) и YACC (*Yet Another Compiler Compiler Compiler*). Эти утилиты получают на вход контекстно-свободную грамматику и производят на выходе код синтаксического анализа, способный токенизировать и анализировать программы, написанные на этой грамматике. Затем сгенерированный код можно настраивать в соответствии с конкретными потребностями автора компилятора. Однако, следуя духу практического курса «От Nand до “Тетриса”», мы решили не использовать такие «черные ящики» для реализации нашего компилятора, а построить все с нуля.

11. Компилятор II: генерация кода

Работая над проблемой, я никогда не задумываюсь о красоте. Но если под конец получается некрасивое решение, я понимаю, что оно неправильное.

— Р. Бакминстер Фуллер (1895–1983)

Большинство программистов воспринимают компиляторы как нечто само собой разумеющееся. Но если задуматься, перевод программы высокого уровня в двоичный код — это почти магия. И раскрытию этой магии мы посвятили целых четыре главы (7–11) курса «От Nand до “Тетриса”». Наша практическая методология основана на разработке компилятора для языка Jack — простого, современного, объектно-ориентированного языка программирования. Как и в случае с Java и C#, компилятор Jack в целом строится на двух уровнях: на виртуальной машине (ВМ), служащей своего рода «бэкендом», переводящим команды ВМ на машинный язык, и на компиляторе, служащем «фронтендом» и переводящем программы Jack в код ВМ. Создание компилятора — сложная задача, поэтому мы разделили его на два концептуальных модуля: *синтаксический анализатор*, разрабатываемый в главе 10, и генератор кода — предмет данной главы.

Синтаксический анализатор создавался для того, чтобы продемонстрировать возможность анализа высокоуровневых программ и их разбора (парсинга) на лежащие в их основе синтаксические элементы. В этой главе мы превратим анализатор в полноценный компилятор — программу, которая преобразует разобранные элементы в команды

ВМ, предназначенные для выполнения на абстрактной виртуальной машине, описанной в главах 7–8. Этот подход следует модульной парадигме анализа — синтеза, лежащей в основе построения хорошо продуманных компиляторов. Он также отражает саму суть трансляции, или перевода текста с одного языка на другой: сначала для анализа исходного текста и выяснения его глубинной *семантики* (то есть того, что хотят сказать этим текстом) используется *синтаксис* исходного языка; затем эта семантика заново выражается с помощью синтаксиса целевого языка. В контексте данной главы исходный язык и целевой язык — это, соответственно, Jask и язык ВМ.

Современные языки программирования высокого уровня весьма богаты и обладают многочисленными средствами выражения. Они позволяют создавать и использовать сложные абстракции, такие как функции и объекты, выражать алгоритмы с помощью элегантных высказываний и создавать структуры данных неограниченной сложности. В отличие от них аппаратные платформы, на которых в конечном счете и выполняются эти программы, отличаются лаконичностью и минимализмом. Как правило, они предлагают не более чем некий набор регистров для хранения данных и некий набор примитивных инструкций для их обработки. Таким образом, перевод программ с высокого уровня на низкий — это довольно сложная задача. Но если целевая платформа — это виртуальная машина, а не какая-то «голая» аппаратная платформа, то задача несколько облегчается, поскольку абстрактные команды виртуальной машины не настолько примитивны, как конкретные машинные инструкции. Тем не менее разрыв между выразительным языком высокого уровня и языком виртуальных машин велик и сложен для преодоления.

Глава начинается с общего обсуждения *генерации кода*, поделенного на шесть разделов. Сначала мы рассказываем, как компиляторы используют *таблицы символов* для привязки символьных переменных к сегментам виртуальной памяти. Далее мы описываем алгоритмы компиляции *выражений* и *строк* символов. Затем представлены методы компиляции *высказываний* типа *let*, *if*, *while*, *do* и *return*. В целом методы компиляции переменных, выражений и высказываний создают основу разработки компиляторов для простых, процедурных

и C-подобных языков. В оставшейся части раздела 11.1 мы обсудим компиляцию *объектов* и *массивов*.

Раздел 11.2 («Спецификация») содержит рекомендации по отображению программ на языке Jack на платформе и языке VM, а раздел 11.3 («Реализация») описывает архитектуру программного обеспечения и API для завершения разработки компилятора. Как обычно, глава заканчивается разделом «Проект», содержащим пошаговые инструкции и описание тестовых программ для завершения создания компилятора, и разделом «Перспектива», в котором затрагиваются различные моменты, оставшиеся за рамками главы.

Так в чем же польза этой главы лично для вас? Понять, как устроены и работают компиляторы, желают многие специалисты, хотя немногие из них в итоге решаются поработать своими руками над созданием его с нуля. Так получается, потому что цена данного опыта — по крайней мере, в академической сфере — обычно представляет собой посещение сложного факультативного курса, рассчитанного на целый семестр. В нашем практическом курсе «От Nand до “Тетриса”» ключевые элементы этого процесса рассматриваются в четырех главах и проектах, кульминацией которых служит эта глава. В процессе работы мы обсуждаем и иллюстрируем ключевые алгоритмы, структуры данных и приемы программирования, лежащие в основе построения типичных компиляторов. Понаблюдав за тем, как эти умные идеи и приемы используются на практике, можно в очередной раз подивиться тому, как человеческая изобретательность превращает примитивный механизм из переключателей и логических элементов в нечто, приближающееся к волшебству.

11.1. Генерация кода

Программисты высокого уровня работают с абстрактными строительными блоками, такими как *переменные*, *выражения*, *высказывания (операции)*, *процедуры*, *объекты* и *массивы*. С помощью этих абстрактных строительных блоков программисты описывают некие действия создаваемой ими программы. Задача компилятора — перевести семантику этих действий на язык, понятный целевому компьютеру.

В нашем случае целевой компьютер — это виртуальная машина, описанная в главах 7–8. Таким образом, нам нужно понять, как систематически переводить *выражения, высказывания, процедуры* и действия с *переменными, объектами и массивами* в последовательность команд стековой ВМ, выполняющих семантику исходной программы на целевой виртуальной машине. Нам не нужно беспокоиться о последующем переводе программ ВМ на машинный язык, поскольку мы уже позаботились об этой дополнительной сложности в проектах 7–8 благодаря принципам двухуровневой компиляции и модульного дизайна.

```

/** Представляет точки на двумерной плоскости.
    File name: Point.jack. */
class Point {
    // Координаты этой точки:
    field int x, y

    // Количество до сих пор созданных объектов Point:
    static int pointCount;

    /** Создает точку на двумерной плоскости
        и инициализирует ее с заданными координатами. */
    constructor Point new(int ax, int ay) {
        let x \ ax;
        let y \ ay;
        let pointCount = pointCount + 1;
        return this;
    }

    /** Возвращает координату x этой точки. */
    method int getX() { return x; }

    /** Возвращает координату y этой точки. */
    method int getY() { return y; }

    /** Возвращает количество до сих пор созданных
        объектов Point. */
    function int getPointCount() {
        return pointCount;
    }
}

// Объявление класса продолжается сверху справа.

```

```

/** Возвращает точку, являющуюся суммой
    этой и другой. */
method Point plus(Point other) {
    return Point.new(x + other.getX(),
                    y + other.getY());
}

/** Возвращает евклидово расстояние между
    этой точкой и другой. */
method int distance(Point other) {
    var int dx, dy;
    let dx \ x - other.getX();
    let dy \ y - other.getY();
    return Math.sqrt((dx*dx) + (dy*dy));
}

/** Печатает эту точку как "(x, y)" */
method void print() {
    do Output.printString("(");
    do Output.printInt(x);
    do Output.printString(",");
    do Output.printInt(y);
    do Output.printString(")");
    return;
}

} // Конец объявления класса Point.

```

Иллюстрация 11.1. Класс Point. В этом классе представлены все возможные типы переменных (field, static, local и argument) и процедур (constructor, method и function), а также процедуры, возвращающие примитивные типы, типы объектов и процедуры типа void. Здесь же показаны вызовы функций, конструкторов и методов для текущего объекта (this) и других объектов.

На протяжении всей главы мы приводим примеры компиляции различных частей класса `Point`, представленного ранее в этой книге. Объявление класса повторяется на иллюстрации 11.1, демонстрирующей большинство возможностей языка Jack. Советуем вам сейчас бегло просмотреть этот код, чтобы освежить в памяти функциональность класса `Point`. После этого вы будете готовы погрузиться в увлекательное путешествие по систематическому сокращению этой высокоуровневой функциональности и любой другой подобной объектно-ориентированной программы в сжатый код ВМ.

11.1.1. Обработка переменных

Одна из основных задач компиляторов — привязка переменных, объявленных в исходной программе высокого уровня, к оперативной памяти целевой платформы (отображение переменных в памяти). Рассмотрим для примера язык Java: переменные `int` предназначены для представления 32-битных значений; переменные `long` для представления 64-битных значений и т. д. Если оперативная память хоста имеет ширину в 32 бита, компилятор привяжет переменные `int` и `long` соответственно к одному слову и к двум последовательным словам памяти. В практическом курсе «От Nand до “Тетриса”» сложностей с отображением нет: все примитивные типы в языке Jack (`int`, `char` и `boolean`) имеют ширину в 16 бит, так же как и адреса и слова оперативной памяти Hack. Таким образом, каждую переменную Jack, включая переменные-указатели, содержащие 16-битные адресные значения, можно привязать ровно к одному слову памяти.

Вторая проблема, с которой сталкиваются разработчики компиляторов, заключается в том, что переменные разных *типов* имеют разные жизненные циклы. Статические переменные уровня класса разделяются глобально всеми процедурами класса. Поэтому одна копия каждой статической переменной должна сохраняться в течение всего времени выполнения программы. С *полевыми* (`field`) переменными уровня экземпляра поступают иначе: каждый объект (экземпляр класса) должен иметь частный набор своих полевых переменных, и, когда объект больше не нужен, выделенную для них память следует освободить.

Локальные переменные и переменные аргументов на уровне процедуры создаются каждый раз при запуске процедуры, и выделенную для них память нужно освобождать при ее завершении.

Это плохая новость. Хорошая новость заключается в том, что мы уже справились со всеми этими трудностями. В нашей двухуровневой архитектуре компилятора задачи по выделению и освобождению памяти перенесены на уровень виртуальной машины. Все, что нам теперь нужно сделать, это связать *статические* переменные Jack с переменными `static 0, static 1, static 2...`; *полевые* переменные `c this 0, this 1...`; *локальные* переменные `c local 0, local 1...`; и *переменные аргументов* `c argument 0, argument 1...` Последующее отображение сегментов виртуальной памяти на оперативную память хоста, а также сложное управление их жизненными циклами во время выполнения полностью берет на себя реализация ВМ.

Напомним, что эта реализация далась не так уж легко: нам пришлось потрудиться над созданием ассемблерного кода, динамически отображающего сегменты виртуальной памяти на оперативную память хоста в качестве побочного эффекта реализации протокола вызова и возврата функций. Теперь мы можем пожинать плоды этих усилий: единственное, что требуется от компилятора, — это связать переменные высокого уровня с сегментами виртуальной памяти. Все последующие операции, связанные с управлением этими сегментами в оперативной памяти, будут обрабатываться реализацией виртуальной машины. Вот почему мы иногда называем реализацию ВМ «бэкендом» компилятора.

Напомним, что в двухуровневой модели компиляции работу с переменными можно свести к отображению переменных высокого уровня на сегменты виртуальной памяти и использованию этого отображения по мере необходимости при генерации кода. Этими задачами можно легко управлять с помощью такой классической абстракции, как *таблица символов*.

Таблица символов: когда компилятор встречает переменные в высокоуровневом выражении, таком как, например, `let y = foo(x)`, ему нужно как-то узнавать, что они обозначают. Является ли *x*

статической переменной, полем объекта, локальной переменной или аргументом подпрограммы? Представляет ли эта переменная целое число `int`, булево значение `boolean`, символ `char` или какой-либо классовый тип? Для генерации кода на все эти вопросы нужно как-то отвечать каждый раз при появлении в коде переменной *x*. Разумеется, с переменной *y* следует обращаться точно так же.

Удобный инструмент для отслеживания переменных — *таблица символов*. Когда в исходном коде объявляется какая-либо переменная — статическая, полевая, локальная или аргументов, — компилятор присваивает ей следующую доступную запись в соответствующем сегменте ВМ (`static`, `this`, `local` или `argument`) и записывает соответствие в таблицу символов. Когда эта переменная встречается в другом месте кода, компилятор ищет ее имя в таблице символов, извлекает ее свойства и при необходимости использует их для генерации кода.

Важная особенность языков высокого уровня заключается в *разделении пространств имен*: один и тот же идентификатор в разных областях программы может означать разные сущности. Для обеспечения разделения пространств имен каждый идентификатор неявно ассоциируется с *областью видимости*, то есть с той областью программы, в которой он распознается. В языке Jack область видимости статических и полевых переменных — это класс, в котором они объявлены, а область видимости локальных переменных и переменных аргументов — это процедура (подпрограмма), в которой они объявлены. Компиляторы Jack могут реализовать абстракции области видимости, создавая и поддерживая две отдельные таблицы символов, как показано на иллюстрации 11.2.

Области видимости вложены друг в друга, причем внутренние области перекрывают внешние. Например, встречая выражение $x + 17$, компилятор Jack сначала проверяет, является ли *x* переменной уровня процедуры (локальной переменной или переменной аргумента). Если этого не происходит, компилятор проверяет, является ли *x* статической переменной или полем. Некоторые языки поддерживают вложенность неограниченной глубины, позволяя переменным быть локальными в любом блоке кода, в котором они объявлены. Для поддержки

неограниченной вложенности компилятор может использовать связанный список таблиц символов, каждая из которых отражает одну область видимости, вложенную в следующую в списке. Если компилятор не может найти переменную в таблице, связанной с текущей областью видимости, он ищет ее в следующей таблице списка, от внутренних к более наружным. Если переменная не найдена в списке, компилятор выдает ошибку «необъявленная переменная».

В языке Jack существуют только два уровня областей видимости: область процедуры, которая в данный момент компилируется, и область класса, в котором эта подпрограмма объявлена. Поэтому компилятор может обойтись только двумя таблицами символов.

Высокоуровневый (Jack) код

```
class Point {
  field int x, y;
  static int pointCount;
  ...
  method int distance(Point other) {
    var int dx, dy;
    let dx \ x - other.getx();
    let dy \ y - other.gety();
    return Math.sqrt((dx*dx) + (dy*dy));
  }
  ...
}
```

Имя	Тип	Вид	#	Таблица символов уровня класса
x	int	field	0	
y	int	field	1	
pointCount	int	static	0	

Имя	Тип	Вид	#	Таблица символов уровня процедуры
this	Point	arg	0	
other	Point	arg	1	
dx	int	var	0	
dy	int	var	1	

Иллюстрация 11.2. Примеры таблиц символов. О строке `this` в таблице уровня процедуры будет сказано далее в этой главе.

Обработка объявлений переменных: начиная компиляцию объявления класса, компилятор Jack создает таблицу символов уровня класса и таблицу символов уровня процедуры. Разбирая объявление статической или полевой переменной, он добавляет новую строку в таблицу символов уровня класса. В строке записывается *имя* переменной, *тип* (целочисленный, булевый, символьный или имя класса), *вид* (статическая или полевая) и *индекс* внутри вида.

Приступая к компиляции объявления процедуры (конструктора, метода или функции), компилятор Jack обнуляет таблицу символов уровня процедуры. Если процедура является методом, компилятор

добавляет в таблицу символов уровня процедуры строку `<this, className, arg, 0>` (эту подробность инициализации мы опишем в разделе 11.1.5.2, и до тех пор ее можно игнорировать). Приступая к разбору объявления локальной или аргументной переменной, он добавляет в таблицу символов уровня подпрограммы новую строку, записывая имя переменной, тип (целочисленный, булевый, символьный или имя класса), вид (`var` или `arg`) и индекс внутри вида. Индекс каждого вида (`var` или `arg`) начинается с 0 и увеличивается на 1 после каждого добавления в таблицу новой переменной этого вида.

Обработка переменных в высказываниях: встречая переменную в высказывании, компилятор ищет имя переменной в таблице символов уровня процедуры. Если переменная не найдена, компилятор ищет ее в таблице символов уровня класса. Найдя переменную, компилятор завершает трансляцию высказывания. Рассмотрим для примера таблицы символов, показанные на иллюстрации 11.2, и предположим, что мы компилируем высказывание высокого уровня `let y = y + dy`. Компилятор транслирует его в команды ВМ `push this 1, push local 1, add, pop this 1`. Здесь мы предполагаем, что компилятор знает, как работать с выражениями и высказываниями `let`, которые рассматриваются в следующих двух разделах.

11.1.2. Компиляция выражений

Начнем с компиляции простых выражений, таких как $x + y - 7$. Под «простым выражением» мы понимаем последовательность *term operator term operator term...*, где каждый член *term* — либо переменная, либо константа, а каждый *operator* — это `+`, `-`, `*` или `/`.

В Jack, как и в большинстве языков высокого уровня, выражения записываются с использованием *инфиксной* нотации: сложение x и y обозначается как $x + y$. Целевой же язык нашей компиляции — *постфиксный*: та же семантика сложения в коде стековой ВМ выражается как `push x, push y, add`. В главе 10 мы описали алгоритм, раскладывающий исходный инфиксный код на элементы с помощью XML-тегов. Логика этого алгоритма парсинга можно оставить прежней,

но выходную часть следует изменить для генерации постфиксных команд. На иллюстрации 11.3 показано различие этих систем.

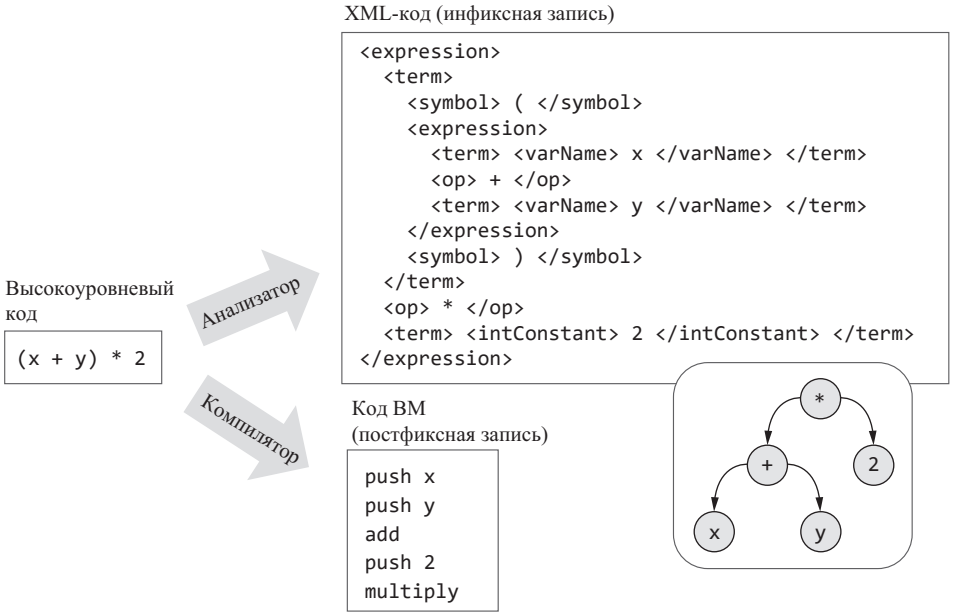


Иллюстрация 11.3. Инфиксный и постфиксный способы передачи одной и той же семантики.

Напомним, что нам нужен алгоритм, умеющий разбирать инфиксное выражение и генерировать из него на выходе постфиксный код, реализующий ту же семантику на стековой машине. На иллюстрации 11.4 представлен один из таких алгоритмов. Алгоритм обрабатывает входное выражение слева направо, генерируя код ВМ по ходу работы. Удобно, что данный алгоритм также обрабатывает унарные операторы и вызовы функций.

Если мы выполним код стековой ВМ, сгенерированный алгоритмом codeWrite (правая часть иллюстрация 11.4), выполнение закончится тем, что будут использованы все члены выражения, а итоговое значение выражения будет размещено на вершине стека. Именно этого и ожидают от скомпилированного кода выражения.

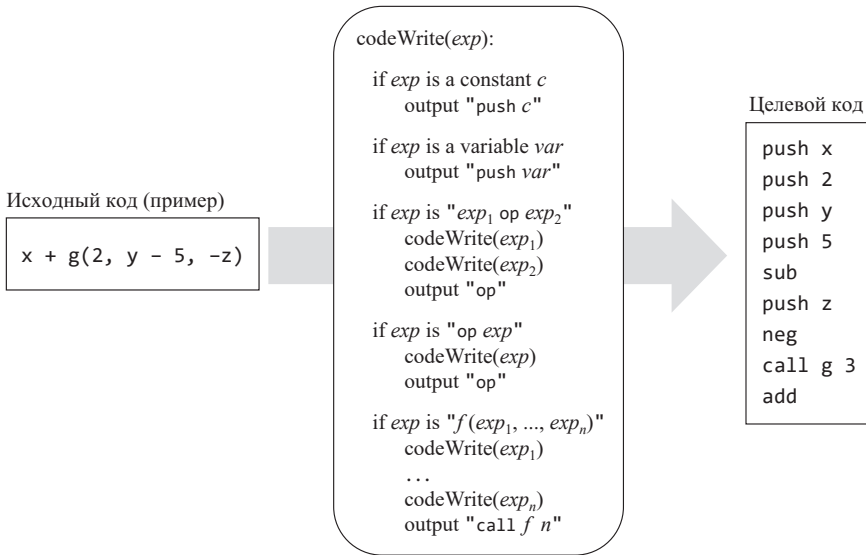


Иллюстрация 11.4. Алгоритм генерации кода ВМ для выражений и пример компиляции. Алгоритм предполагает, что входное выражение допустимое. Окончательная реализация этого алгоритма должна заменить выдаваемые символические переменные на соответствующие им отображения в таблице символов.

До сих пор мы имели дело с относительно простыми выражениями. На иллюстрации 11.5 приведено полное грамматическое определение выражений (*expressions*) Jack, а также несколько примеров реальных выражений, соответствующих этому определению.

Компиляция выражений Jack будет осуществляться программой с именем `compileExpression`. Разработчик этой процедуры должен начать с алгоритма, показанного на иллюстрации 11.4, и расширить его для обработки различных возможностей, указанных на иллюстрации 11.5. Более подробно о данной реализации мы расскажем позже в этой главе.

Определение (из грамматики Jack):

```

expression: term (op term) *

term: integerConstant | stringConstant | keywordConstant | varName |
      varName '[' expression ']' | '(' expression ')' | (unaryOp term) | subroutineCall

subroutineCall: subroutineName '(' expressionList ')' |
                (className | varName) '.' subroutineName '(' expressionList ')'

expressionList: (expression (',' expression) *) ?

op: '+' | '-' | '*' | '/' | '%' | '|' | '<' | '>' | '='

unaryOp: '-' | '~'

keywordConstant: 'true' | 'false' | 'null' | 'this'

```

Определения *integerConstant*, *stringConstant*, *keywordConstant* и других элементов даны в полной грамматике Jack (рис. 10.6) и не требуют пояснений.

Примеры: 5
 x
 x + 5
 (-b + Math.sqrt(b*b - (4 * a * c))) / (2 * a)
 arr[i] + foo(x)
 foo(Math.abs(arr[x + foo(5)]))

Иллюстрация 11.5. Выражения в языке Jack.

11.1.3. Компиляция строк

В компьютерных программах широко используются строки — последовательности символов. Объектно-ориентированные языки обычно работают со строками как с экземплярами класса *String* (класс *String*, входящий в состав ОС Jack, описан в приложении 6). Каждый раз, когда в высокоуровневом операторе или выражении появляется строковая константа, компилятор генерирует код, вызывающий конструктор *String*, создающий и возвращающий новый объект *String*. Затем компилятор инициализирует новый объект символами строки. Для этого создается последовательность вызовов метода *String appendChar*, по одному для каждого символа, перечисленного в строковой константе высокого уровня.

Такую реализацию строковых констант можно назвать расточительной, и она потенциально может привести к утечкам памяти. Рассмотрим

для примера высказывание `Output.println("Loading ... please wait")`. Все, что, по всей видимости, хочет сделать программист на высокоуровневом языке, — это вывести сообщение; его, конечно, не волнует, создаст ли компилятор новый объект, и он может удивиться, узнав, что объект будет храниться в памяти до завершения программы. Но именно так это и произойдет: будет создан новый объект `String`, и этот объект будет храниться в фоновом режиме, ничего больше не делая.

Java, C# и Python используют во время выполнения процесс *сборки мусора*, который освобождает память, занимаемую более неиспользуемыми объектами (технически это объекты, на которые не ссылается никакая переменная). В целом для эффективного использования строковых объектов современные языки программирования используют различные методы оптимизации и специализированные строковые классы. В ОС Jack имеется только один класс `String` и нет никаких оптимизаций, связанных со строками.

Службы операционной системы: в работе со строками мы впервые упомянули, что компилятор может по мере необходимости использовать службы ОС. Действительно, разработчики компиляторов Jack могут считать, что каждый конструктор, метод и функция, перечисленные в OS API (приложение 6), доступны в виде *скомпилированной функции ВМ*. С технической точки зрения любую из этих функций ВМ можно вызвать кодом, сгенерированным компилятором. Полностью такая конфигурация будет реализована в главе 12, в которой мы создадим ОС на языке Jack и скомпилируем ее в код ВМ.

11.1.4. Компиляция высказываний

В языке программирования Jack есть пять высказываний: `let`, `do`, `return`, `if` и `while`. Перейдем к тому, как компилятор Jack генерирует код ВМ, обрабатывающий семантику этих высказываний.

Компиляция высказываний `return`: теперь, когда мы знаем, как компилировать выражения, реализовать компиляцию *выражения* `return`

будет просто. Сначала вызываем процедуру `compileExpression`, которая генерирует код ВМ, предназначенный для вычисления и помещения значения выражения в стек. Затем генерируем команду ВМ `return`.

Компиляция высказываний `let`: здесь мы обсудим обработку высказываний вида `let varName = expression`. Поскольку процесс синтаксического разбора происходит слева направо, мы начинаем с того, что запоминаем `varName`. Затем вызываем процедуру `compileExpression`, которая помещает значение выражения в стек. Наконец, генерируем команду ВМ `pop varName`, где `varName` — это отображение `varName` в таблице символов (например, `local 3`, `static 1` и т. д.).

Компиляцию высказываний вида `let varName [expression1] = expression2` мы обсудим позже, в разделе, посвященном работе с массивами.

Компиляция высказываний `do`: здесь мы обсудим компиляцию вызовов функций вида `do className.functionName(exp1, exp2, ..., expn)`. Абстракция `do` предназначена для вызова подпрограммы с игнорированием ее возвращаемого значения. В главе 10 мы рекомендовали компилировать такие высказывания, как если бы их синтаксис был *выражением* `do`. Здесь мы повторяем эту рекомендацию: для компиляции высказывания `do className.functionName (...)` вызываем `compileExpression`, а затем избавляемся от самого верхнего элемента стека (значения выражения), генерируя команду типа `pop temp 0`.

Компиляцию *вызовов методов* вида `do varName.methodName (...)` и `do methodName (...)` мы обсудим позже, в разделе, посвященном данному вопросу.

Компиляция высказываний `if` и `while`. В языках программирования высокого уровня используется множество *операторов управления потоком*, таких как `if`, `while`, `for` и `switch`, из которых в Jack есть `if` и `while`. В отличие от них низкоуровневые языки ассемблера и ВМ

управляют потоком выполнения программы с помощью двух примитивов ветвления: *условного goto* и *безусловного goto*. Поэтому одна из проблем, с которой сталкиваются разработчики компиляторов, — это передача семантики высокоуровневых операторов управления потоком с использованием одних лишь примитивов *goto*. На иллюстрации 11.6 показано, как можно устранить этот пробел.

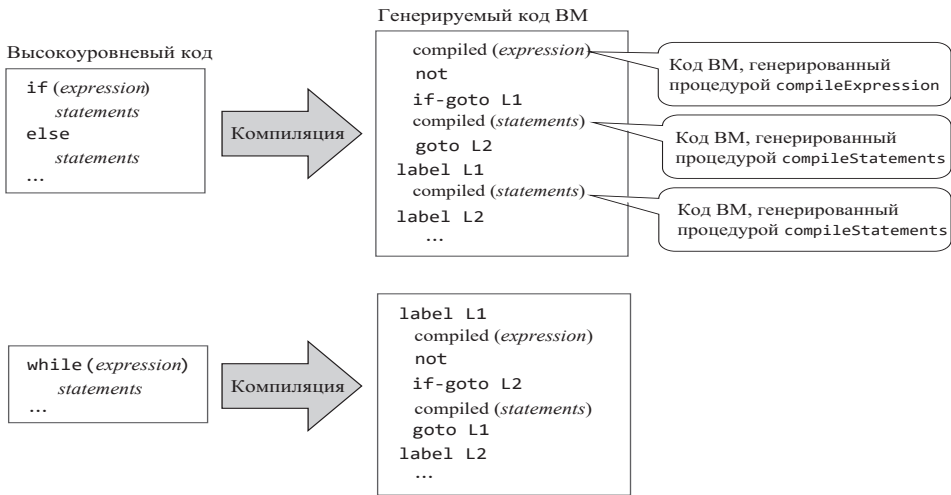


Иллюстрация 11.6. Компиляция высказываний *if* и *while*. Метки *L1* и *L2* генерируются компилятором.

Обнаруживая ключевое слово `if`, компилятор знает, что должен разобрать шаблон вида `if (expression) {statements} else {statements}`. Поэтому начинает с вызова процедуры `compileExpression`, генерирующей команды ВМ, призванные вычислить и поместить значение выражения в стек. Затем компилятор генерирует команду ВМ `not`, инвертирующую значение выражения. Далее он создает метку, допустим, `L1`, и использует ее для генерации команды `if-goto L1`. После этого компилятор вызывает `compileStatements`. Эта процедура предназначена для компиляции последовательности вида `statement; statement; ... statement`, где каждое `statement` — это `let`, `do`, `return`, `if` или `while`. На иллюстрации 11.6 получившийся в результате код ВМ

условно называется *compiled (statements)*. Остальная часть стратегии компиляции не требует пояснений.

В высокоуровневых программах обычно содержатся несколько примеров `if` и `while`. Чтобы справиться с этой сложностью, компилятор может генерировать глобально уникальные метки, например, метки с суффиксом в виде значения работающего счетчика. Кроме того, операторы управления потоком часто бывают вложенными, например, `if` внутри `while`, внутри еще одного `while` и т. д. Такие вложения обрабатываются неявно, поскольку процедура `compileStatements` по своей сути рекурсивна.

11.1.5. Обработка объектов

До сих пор в этой главе мы описывали способы компилирования *переменных, выражений, строк и высказываний*. Это почти все, что нужно знать для создания компилятора для процедурного языка, похожего на C. Однако в практическом проекте «От Nand до “Тетриса”» мы ставим перед собой более высокую цель: создание компилятора для объектно-ориентированного языка, похожего на Java. Поэтому перейдем к работе с *объектами*.

Объектно-ориентированные языки имеют средства для работы с такими абстракциями, как *объекты*. Каждый объект физически реализуется как блок памяти, на который можно ссылаться с помощью статической, полевой, локальной или аргументной переменной. Базовый адрес блока памяти объекта содержит ссылочная переменная, которую также называют *объектной переменной* или *указателем*. Операционная система реализует эту модель, управляя логической областью в оперативной памяти, называемой *кучей*. *Куча* используется как резерв памяти, из которого по мере необходимости вырезаются блоки памяти для представления новых объектов. Когда объект больше не нужен, его блок памяти можно освободить и вернуть в кучу. Компилятор выполняет эти действия по управлению памятью посредством вызова функций ОС, как мы увидим позже.

В любой момент выполнения программы куча может содержать множество объектов. Предположим, что мы хотим к одному

из этих объектов, допустим `p`, применить метод `foo`. В объектно-ориентированном языке это делается с помощью идиомы вызова метода `p.foo()`. Рассмотрим этот момент с точки зрения вызываемой процедуры. Как и любой другой, метод `foo` предназначен для работы с *текущим объектом*, то есть `this`. В частности, когда команды ВМ в коде `foo` ссылаются на `this 0`, `this 1`, `this 2` и т. д., они должны воздействовать на *поля* объекта `p`, для которого был вызван `foo`. В связи с этим возникает вопрос: как сопоставить сегмент `this` с объектом `p`?

Построенная в главах 7–8 виртуальная машина имеет механизм для реализации такого сопоставления: двухзначный сегмент указателя `pointer`, отображаемый непосредственно на участки 3–4 оперативной памяти, также называемые `THIS` и `THAT`. Согласно спецификации ВМ, указатель `THIS` (на который можно ссылаться как на `pointer 0`) предназначен для хранения базового адреса сегмента памяти `this`. Таким образом, чтобы сопоставить сегмент `this` с объектом `p`, мы помещаем (`push`) значение `p` (которое является адресом) в стек, а затем помещаем (`pop`) его в `pointer 0`. Разновидности такого способа инициализации часто используются при компиляции конструкторов и методов, о чем мы сейчас и поговорим.

11.1.5.1. Компиляция конструкторов

В объектно-ориентированных языках объекты создаются процедурами, называемыми *конструкторами*. В этом разделе мы опишем, как компилировать вызов конструктора (например, оператор `new` в Java) с точки зрения *вызывающей* стороны и как компилировать код самого конструктора, то есть *вызываемой* стороны.

Компиляция вызовов конструктора: создание объекта обычно состоит из двух этапов. Сначала объявляется переменная некоторого типа класса, например, `var Point p`. На более поздней стадии создается объект в виде экземпляра этого класса посредством вызова конструктора класса, например, `let p = Point.new(2, 3)`. Или, в зависимости от используемого языка, можно объявить *и одновременно*

конструировать объекты с помощью одного оператора высокого уровня. Однако за кулисами это действие всегда разбивается на два отдельных этапа: объявление и конструкция.

Рассмотрим повнимательнее высказывание `let p = Point.new(2, 3)`. Эту абстракцию можно описать так: «Пусть конструктор `Point.new` выделит двухсловный блок памяти для представления нового экземпляра `Point`, инициализирует два слова этого блока значениями 2 и 3, и пусть `p` ссылается на базовый адрес данного блока». В этой семантике подразумеваются два предположения. Во-первых, конструктор знает, как выделять блок памяти нужного размера. Во-вторых, когда конструктор, будучи процедурой, завершает свое выполнение, он возвращает вызывающей стороне базовый адрес выделенного блока памяти. На иллюстрации 11.7 показана возможная реализация этой абстракции.

По поводу иллюстрации 11.7 следует сделать три замечания. Во-первых, обратите внимание, что в компиляции высказываний типа `let p = Point.new(2, 3)` и `let p = Point.new(5, 7)` нет ничего особенного. Мы уже рассуждали о том, как компилировать высказывания `let` и вызовы процедур. Единственное, что делает эти вызовы особенными, — хитроумное предположение, что каким-то образом будут созданы два объекта. Реализация этой магии полностью возложена на компиляцию вызываемой стороны — конструктора. В результате этой магии конструктор создает два объекта, показанные на схеме ОЗУ (RAM) в иллюстрации 11.7. Это подводит нас ко второму наблюдению: конкретные физические адреса 6012 и 9543 не имеют значения; высокоуровневый код, а также скомпилированный код ВМ не имеют ни малейшего представления о том, где именно в памяти хранятся объекты; ссылки на эти объекты исключительно символические: `p1` и `p2` в высокоуровневом коде и `local 0` и `local 1` в скомпилированном коде. Мимоходом заметим, что это делает программу переносимой и более безопасной. В-третьих, констатируем очевидное, что до тех пор, пока не будет выполнен сгенерированный код ВМ, ничего существенного не происходит. В частности, во время *процесса компиляции* обновляется таблица символов, генерируется низкоуровневый код — и все. Объект будет сконструирован и привязан к переменным

только во время *выполнения* низкоуровневого, то есть скомпилированного кода.

Высокоуровневый код

```
// Может появляться в любом классе:
...
// Объявляет три локальные переменные:
var Point p1, p2;
var int d;
...
// Создает два объекта:
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```



Generated VM code

```
...
// var Point p1, p2;
// var int d;
// Компилятор добавляет p1, p2 и d к таблице
// символов процедуры, код не генерируется.
...
// let p1 = Point.new(2,3);
// Для обработки правой части компилятор вызывает
// compileExpression. В результате в вершину стека
// помещаются значения 2 и 3 и генерируется
// команда call Point.new 2.
// Компилятор генерирует команду:
pop local 0
// let p2 = Point.new(5,7);
// Аналогично.
...
```

Имя	Тип	Вид	#
p1	Point	local	0
p2	Point	local	1
d	int	local	2



Иллюстрация 11.7. Конструирование объекта с точки зрения вызывающей стороны. В этом примере вызывающая сторона объявляет две переменные объекта, а затем вызывает класс конструктора для конструирования этих объектов. Конструктор объектов «магическим» образом выделяет блоки памяти для их представления. Затем код вызова связывает эти две переменные объекта с этими блоками памяти.

Компиляция конструкторов: до сих пор мы рассматривали конструкторы как абстракции «черного ящика»: просто предполагали, что они каким-то образом создают объекты. Иллюстрация 11.8 раскрывает эту магию. Перед ее изучением обратите внимание, что конструктор — это прежде всего *процедура*. Она может иметь аргументы, локальные

переменные и ряд высказываний; поэтому компилятор и рассматривает ее как процедуру. Особенность компиляции конструктора состоит в том, что, помимо обращения с ним как с обычной процедурой, компилятор также должен заодно сгенерировать код, который: 1) создает новый объект и 2) делает новый объект *текущим объектом* (*this*), то есть таковым, над которым будет работать код конструктора.

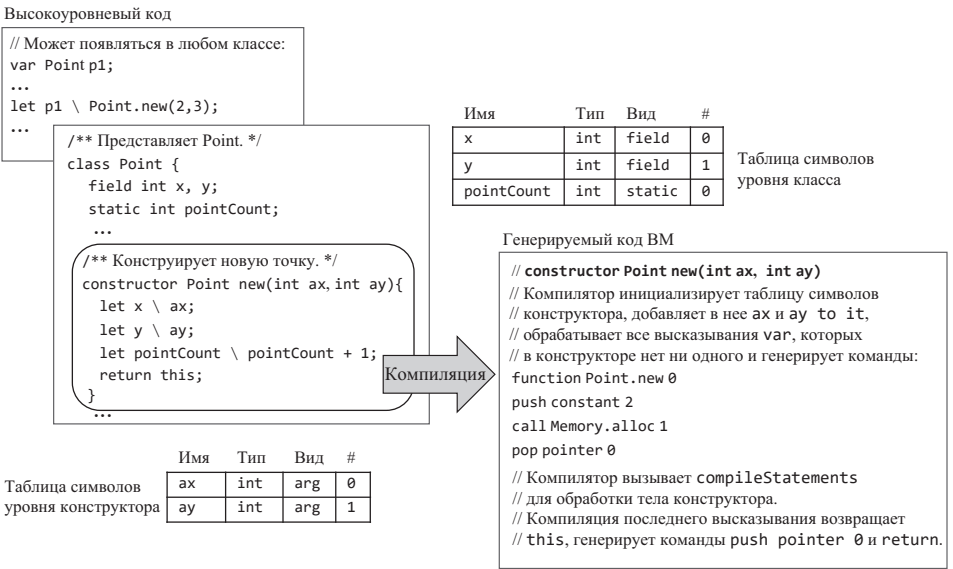


Иллюстрация 11.8. Конструирование объекта с точки зрения конструктора.

При создании нового объекта нужно найти свободный блок оперативной памяти достаточного размера для размещения данных нового объекта и пометить его как используемый. Эти задачи возлагаются на операционную систему хоста. Согласно API ОС, приведенному в приложении 6, функция ОС `Memory.alloc (size)` умеет находить свободный блок оперативной памяти заданного размера *size* (количество 16-битных слов) и возвращать базовый адрес блока.

`Memory.alloc` и родственная ей функция `Memory.deAlloc` используют умные алгоритмы для эффективного выделения и освобождения ресурсов оперативной памяти. Эти алгоритмы будут представлены и реализованы в главе 12, когда мы будем создавать

операционную систему. Пока же достаточно сказать, что компиляторы генерируют низкоуровневый код, который абстрактно использует `alloc` (в конструкторах) и `deAlloc` (в деструкторах).

Перед вызовом `Memory.alloc` компилятор определяет размер необходимого блока памяти. Его можно легко вычислить из таблицы символов уровня класса. Например, таблица символов класса `Point` показывает, что каждый объект `Point` характеризуется двумя значениями `int` (координатами x и y точки). Так компилятор генерирует команды `push constant 2` и `call Memory.alloc 1`, что приводит к вызову функции `Memory.alloc(2)`. Функция ОС `alloc` начинает работу, находит доступный блок ОЗУ размером 2 и помещает его базовый адрес в стек — эквивалент возврата значения для виртуальной машины. Следующее сгенерированное высказывание ВМ — `pop pointer 0` — устанавливает для `THIS` базовый адрес, который возвращает `alloc`. С этого момента сегмент конструктора `this` будет связан с блоком оперативной памяти, выделенным для представления вновь созданного объекта.

Выделив этот сегмент памяти, мы можем легко приступить к генерации кода. Например, когда для обработки высказывания `let x = ax` вызывается процедура `compileLet`, она просматривает таблицы символов, преобразуя `x` в `this 0` и `ax` в `argument 0`. Так `compileLet` генерирует команды `push argument 0` с последующей `pop this 0`. Последняя команда основывается на предположении, что этот сегмент правильно сопоставлен с базовым адресом нового объекта, и мы действительно сделали это, установив для `pointer 0` (на самом деле `THIS`) базовый адрес, возвращенный функцией `alloc`. Эта однократная инициализация гарантирует, что все последующие операции `push / pop this i` выберут правильные адреса в ОЗУ (точнее, в куче). Надеемся, что от читателя не ускользнула замысловатая красота такого решения.

Согласно спецификации языка `Jack`, каждый конструктор должен заканчиваться оператором `return this`. Как следствие, компилятор завершает скомпилированную версию конструктора командами `push pointer 0` и `return`. Эти команды помещают в стек значение `THIS`, базовый адрес конструируемого объекта. В некоторых языках, например в `Java`, конструкторы не обязательно должны завершаться явным

оператором `return this`. Тем не менее скомпилированный код конструкторов Java на уровне виртуальной машины выполняет точно такое же действие, поскольку именно этого и ожидают от конструкторов: создать объект и вернуть его обработку вызывающей стороне.

Вспомним, что только что описанное сложное низкоуровневое действие было вызвано высказыванием `let varName = className.constructorName (...)`. Сейчас понятно, что по замыслу после завершения работы конструктора `varName` в конечном счете сохраняет базовый адрес нового объекта. Под словами «по замыслу» имеется в виду синтаксис высокоуровневой идиомы конструирования объектов и та громоздкая работа, которую должны проделать компилятор, операционная система, транслятор ВМ и ассемблер для реализации этой абстракции. В итоге программисты, пишущие программы на языке высокого уровня, избавляются от всех нудных подробностей построения объектов и могут создавать объекты легко и прозрачно.

11.1.5.2. Компиляция методов

Как и в случае с конструкторами, мы опишем, как компилировать вызовы методов, а затем — как компилировать сами методы.

Компиляция вызовов методов: предположим, мы хотим вычислить евклидово расстояние между двумя точками `p1` и `p2` на плоскости. В процедурном языке в стиле C это можно было бы реализовать с помощью вызова функции типа `distance(p1, p2)`, где `p1` и `p2` представляют составные типы данных. В объектно-ориентированном же языке `p1` и `p2` будут реализованы как экземпляры некоторого класса `Point` и то же самое вычисление будет выполнено с помощью вызова метода типа `p1.distance(p2)`. В отличие от функций, *методы* — это процедуры, работающие всегда с определенным объектом, и ответственность за указание этого объекта лежит на вызывающей стороне. (Тот факт, что здесь метод `distance` принимает в качестве аргумента другой объект `Point`, — случайное совпадение. Вообще говоря, хотя метод и предназначается всегда для работы с объектом, он может иметь 0, 1 или более аргументов любого типа.)

Обратите внимание, что `distance` можно описать как *процедуру* вычисления расстояния от данной точки до другой, а `p1` можно описать как данные, над которыми работает процедура. Также обратите внимание, что обе идиомы `distance(p1, p2)` и `p1.distance(p2)` предназначены для вычисления и возврата одного и того же значения. Однако если в синтаксисе языка *C* основное внимание уделяется расстоянию, то в объектно-ориентированном синтаксисе объект стоит на первом месте в буквальном смысле этого слова. Именно поэтому языки типа *C* иногда называют *процедурными*, а объектно-ориентированные языки называют *управляемыми данными*. Кроме всего прочего, объектно-ориентированный стиль программирования основан на предположении, что объекты знают, как позаботиться о себе. Например, объект `Point` знает, как вычислять расстояние между собой и другим объектом `Point`. Иначе говоря, операция `distance` *инкапсулирована* в определение объекта `Point`.

Агент, ответственный за практическое воплощение всех этих причудливых абстракций, — это, как обычно, трудолюбивый компилятор. Поскольку в целевом языке *ВМ* нет понятия объектов или методов, компилятор обрабатывает объектно-ориентированные вызовы методов, такие как `p1.distance(p2)`, как если бы это были процедурные вызовы вроде `distance(p1, p2)`. Точнее, он переводит `p1.distance(p2)` в команды `push p1, push p2, call distance`. Итак, обобщим: в *Jack* существуют два вида вызова методов:

//Применяет метод к объекту, на который ссылается *varName*:

varName.methodName(*exp*₁, *exp*₂, ..., *exp*_{*n*}).

//Применяет метод к текущему объекту:

methodName(*exp*₁, *exp*₂, ..., *exp*_{*n*}).

//То же, что и *this.methodName*(*exp*₁, *exp*₂, ..., *exp*_{*n*}).

Компиляция метода *varName.methodName*(*exp*₁, *exp*₂, ..., *exp*_{*n*}) начинается с генерации команды `push varName`, где *varName* — отображение *varName* в таблице символов. Если в вызове метода

не упоминается *varName*, в стек помещается отображение *this*. Далее вызывается процедура *compileExpressionList*, вызывающая *compileExpression* *n* раз, по одному разу для каждого выражения в круглых скобках. Наконец, генерируется команда *call className.methodName n + 1*, сообщающая, что в стек было помещено *n + 1* аргументов. Частный случай вызова метода без аргументов транслируется в *call className.methodName 1*. Обратите внимание, что *className* — это тип (*type*) таблицы символов идентификатора *varName*. Пример показан на иллюстрации 11.9.

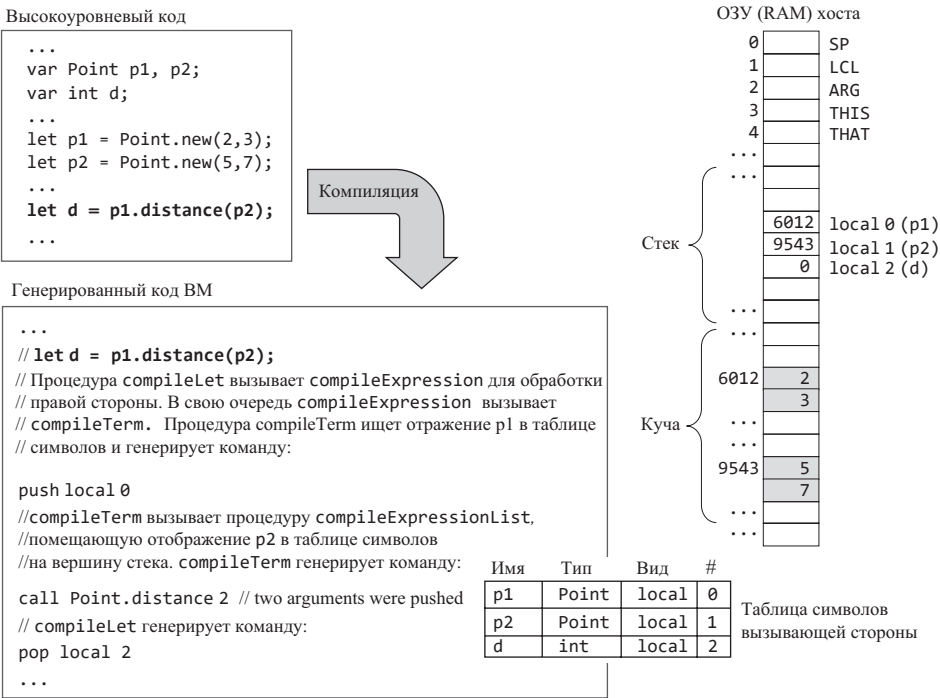


Иллюстрация 11.9. Компиляция вызова метода с точки зрения вызывающей стороны.

Компиляция методов: до сих пор мы обсуждали метод *distance* абстрактно, с точки зрения вызывающей стороны. Рассмотрим, как этот метод можно реализовать, допустим, на языке Java.

```
/** Метод класса Point: возвращает расстояние между этой точкой  
(Point) и другой (other). */  
int distance(Point other) {  
    int dx, dy;  
    dx = x - other.x;  
    dy = y - other.x;  
    return Math.sqrt((dx*dx) + (dy*dy));  
}
```

Как и любой метод, `distance` предназначен для работы с *текущим объектом*, представленным в Java (и в Jack) встроенным идентификатором `this`. Однако, как видно из приведенного выше примера, можно написать целый метод, ни разу не упомянув `this`. Это потому, что дружественный компилятор Java обрабатывает высказывания типа `dx = x - other.x` так, как если бы они были `dx = this.x - other.x`. Такое соглашение делает код высокого уровня более читаемым и простым в написании.

Впрочем, заметим мимоходом, что в языке Jack идиома *object.field* не поддерживается. Как следствие, полями объектов за исключением текущего можно манипулировать только с помощью методов доступа (аксессоров) и методов-модификаторов. Например, выражения типа `x - other.x` реализованы в Jack как `x - other.getx()`, где `getx` — метод доступа в классе `Point`.

Как же компилятор Jack обрабатывает выражения типа `x - other.getx()`? Как и компилятор Java, он ищет `x` в таблицах символов и обнаруживает, что он представляет первое поле в текущем объекте. Но *какой именно* объект из множества объектов представляет *текущий объект*? Согласно алгоритму вызова метода, это должен быть первый аргумент, переданный вызывающей стороной. Поэтому с точки зрения вызываемой стороны текущий объект — это объект, базовый адрес которого представляет собой значение `argument 0`. Таков вкратце низкоуровневый трюк компиляции, реализующий распространенную абстракцию «применить метод к объекту» в языках вроде Java, Python и, конечно же, Jack. Подробности показаны на иллюстрации 11.10.

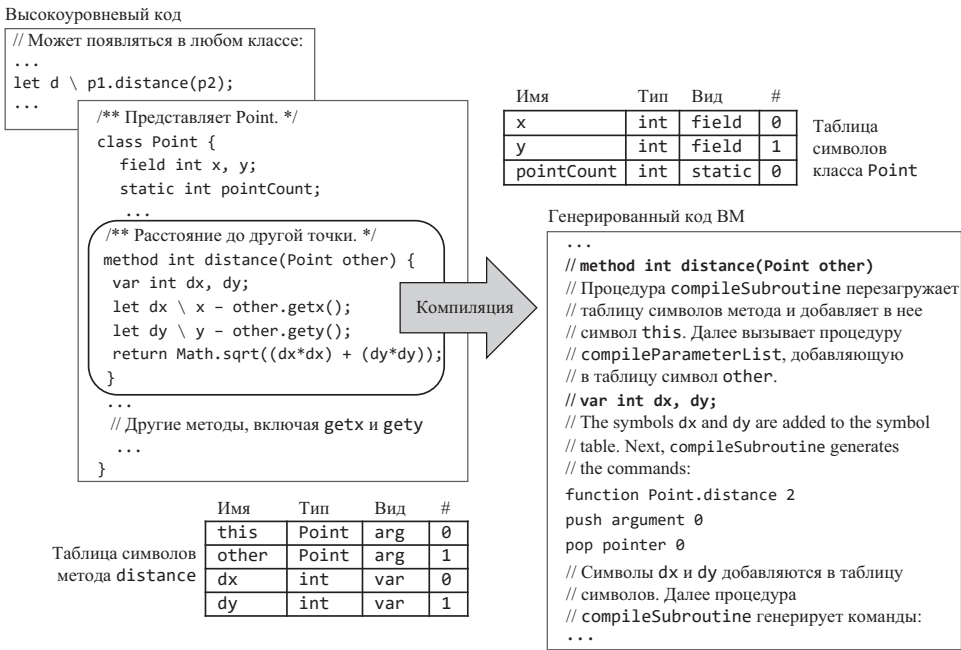


Иллюстрация 11.10. Компиляция методов с точки зрения вызываемой стороны.

Пример начинается в левом верхнем углу иллюстрации 11.10, где код вызывающей стороны выполняет вызов метода `p1.distance(p2)`. Обратим внимание на скомпилированную версию вызываемой стороны и отметим, что собственно код начинается с команды `push argument 0`, за которой следует `pop pointer 0`. Эти команды присваивают указателю `THIS` метода значение аргумента `argument 0`, согласно алгоритму вызова метода, содержащего базовый адрес объекта, для работы над которым и был вызван метод. Таким образом, начиная с этого момента, сегмент `this` метода правильно соотносится с базовым адресом целевого объекта, благодаря чему каждая команда `push / pop this i` также правильно сопоставляется с адресами памяти. Например, высказывание `x = other.getx()` компилируется в последовательность команд `push this 0, push argument 1, call Point.getx 1, sub`. Поскольку мы начали компиляцию кода метода с установки в `THIS` базового адреса вызываемого объекта,

this 0 (и любая другая ссылка this i) гарантированно указывает на поле нужного объекта.

11.1.6. Компиляция массивов

Массивы схожи с объектами. В Jack массивы реализуются как экземпляры класса Array, входящего в состав операционной системы. Таким образом, массивы и объекты объявляются, реализуются и хранятся совершенно одинаково; по сути, массивы — это объекты, с той разницей, что абстракция массива позволяет обращаться к элементам массива с помощью индекса, допустим, let arr[3] = 17. Конкретной эту полезную абстракцию делает компилятор, о чем мы сейчас и поговорим.

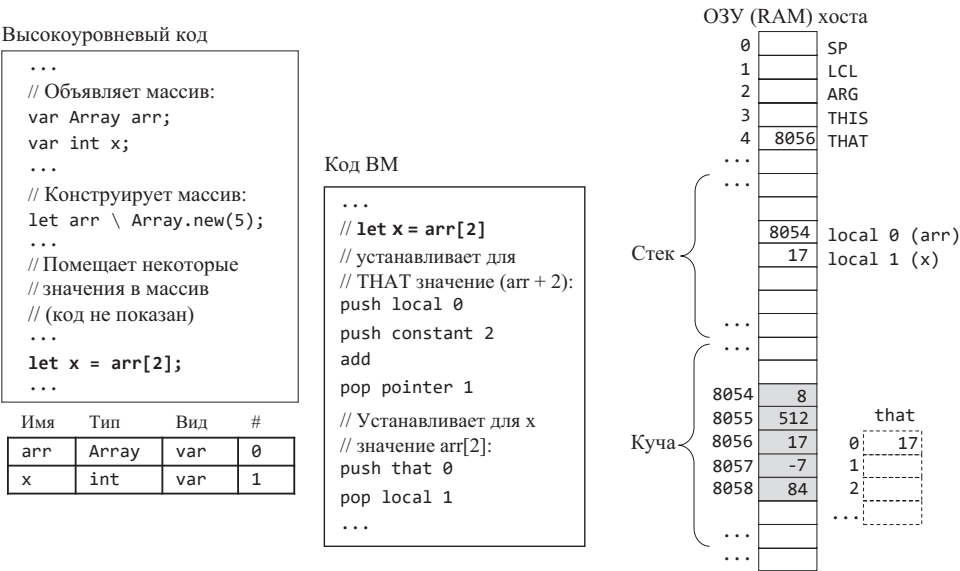


Иллюстрация 11.11. Доступ к массиву с помощью команд VM.

Вспомним, что с использованием нотации указателей arr[i] можно записать как *(arr + i), то есть адрес памяти arr + i. Это ключевая концепция для компиляции высказываний типа let x = arr[i]. Для вычисления физического адреса arr[i] выполняются

команды `push arr`, `push i`, `add`, в результате чего целевой адрес попадает в стек. Затем выполняется `pop pointer 1`. Согласно спецификации ВМ, это действие сохраняет целевой адрес в указателе `THAT` метода (`RAM[4]`), в результате чего базовый адрес виртуального сегмента `that` сопоставляется с целевым адресом. Теперь можно выполнить `push that 0` и `pop x`, на чем и завершается трансляция `let x = arr[i]`. Подробности показаны на иллюстрации 11.11.

Такая стратегия компиляции обладает лишь одним недостатком: она не работает. Точнее, правильнее сказать, что она работает с высказываниями вроде `let a = b[j]`, но не работает, если индексирована левая сторона — как, например, в высказывании `let a[i] = b[j]` на иллюстрации 11.12.

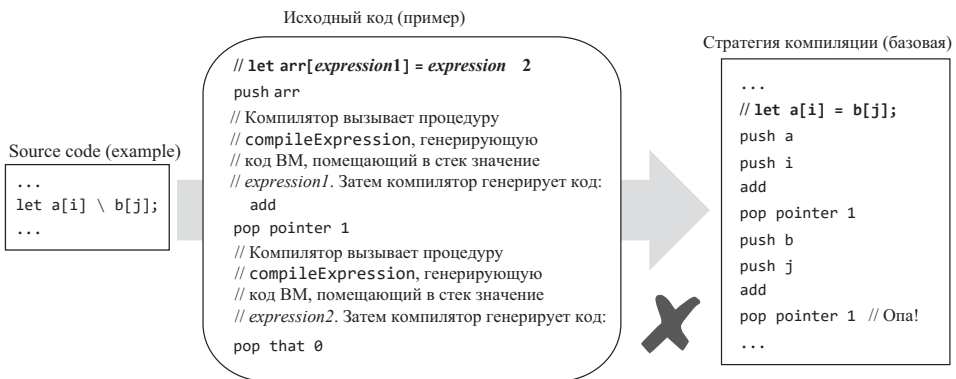


Иллюстрация 11.12. Базовая стратегия компиляции массивов и пример неверного кода, который она может сгенерировать. В данном случае значение, хранимое в `pointer 1`, перезаписывается и адрес `a[i]` теряется.

Хорошая новость заключается в том, что эту ошибочную стратегию компиляции можно легко исправить, чтобы правильно скомпилировать любой экземпляр `let arr[expression1] = expression2`. Как и раньше, начнем с генерации команды `push arr`, вызова процедуры `compileExpression` и генерации команды `add`. Эта последовательность помещает целевой адрес (`arr + expression1`) на вершину стека. Далее мы вызываем процедуру `compileExpression`, которая в итоге помещает на вершину стека значение `expression2`. В данный

момент мы сохраняем значение — это можно сделать с помощью команды `pop temp 0`. Такая операция имеет и приятный побочный эффект — она делает $(arr + expression1)$ верхним элементом стека. Таким образом, теперь можно прописать команды `pop pointer 1`, `push temp 0` и `pop that 0`. Это небольшое исправление вместе с рекурсивной природой процедуры `compileExpression` позволяет обрабатывать высказывания `let arr[expression1] = expression2` любой рекурсивной сложности, такие как, допустим, `let a[b[i] + a[j + b[a[3]]]] = b[b[j] + 2]`.

В заключение следует отметить, что относительно простой компиляцию массивов Jack делают несколько факторов. Во-первых, массивы Jack не типизированы; они скорее предназначены для хранения 16-битных значений без каких-либо ограничений. Во-вторых, все примитивные типы данных в Jack имеют ширину 16 бит, все адреса имеют ширину 16 бит, как и ширина слова оперативной памяти. В сильно типизированных языках программирования и в языках, не гарантирующих такого точного соответствия, компиляция массивов требует дополнительной работы.

11.2. Спецификация

Проблемы компиляции и решения, которые мы описывали до сих пор, можно обобщить для поддержки компиляции любого объектно-ориентированного языка программирования. Перейдем теперь от общего к конкретному: начиная с этого момента и до конца главы мы описываем компилятор Jack. Компилятор Jack — это программа, которая получает на вход программу Jack и генерирует на выходе исполняемый код VM. Код VM реализует семантику программы на виртуальной машине, описанной в главах 7–8.

Использование: компилятор принимает один аргумент командной строки, как показано ниже:

```
prompt> JackCompiler source,
```

где *source* — это либо имя файла вида *Xxx.jack* (расширение обязательно), либо имя папки (в этом случае расширение отсутствует), содержащей один или несколько файлов *.jack*. Имя файла/папки может содержать путь к файлу. Если путь не указан, компилятор работает с текущей папкой. Для каждого файла *Xxx.jack* компилятор создает выходной файл *Xxx.vm* и записывает в него команды ВМ. Выходной файл создается в той же папке, что и входной файл. Если в папке уже есть файл с таким именем, он будет перезаписан.

11.3. Реализация

Перейдем теперь к рекомендациям, советам по реализации и предлагаемому API для расширения синтаксического анализатора, созданного в главе 10, до полноценного компилятора Jack.

11.3.1. Стандартное отображение на виртуальной машине

Компиляторы Jack можно разрабатывать для различных целевых платформ. В этом разделе приводятся рекомендации по отображению различных конструкций языка Jack на одной конкретной платформе, описанной в главах 7–8 виртуальной машины.

Именованние файлов и функций

- Файл класса Jack *Xxx.jack* компилируется в файл класса ВМ с именем *Xxx.vm*.
- Процедура *ууу* в файле *Xxx.jack* компилируется в функцию ВМ с именем *Xxx.ууу*.

Отображение переменных

- Первая, вторая, третья... *статическая* переменная, объявленная в объявлении класса, отображается на виртуальный сегмент памяти `static 0, static 1, static 2...`

- Первая, вторая, третья... *полевая* переменная, объявленная в объявлении класса, отображается на сегмент памяти `this 0, this 1, this 2...`
- Первая, вторая, третья... *локальная* переменная, объявленная в вызываниях `var` процедуры, отображается на сегмент `local 0, local 1, local 2...`
- Первая, вторая, третья... *аргументная* переменная, объявленная в списке параметров функции или конструктора (но не метода), отображается на сегмент `argument 0, argument 1, argument 2...`
- Первая, вторая, третья... *аргументная* переменная, объявленная в списке параметров метода, отображается на сегмент `argument 1, argument 2, argument 3...`

Отображение полей объекта

Чтобы сопоставить виртуальный сегмент `this` с объектом, переданным вызывающей процедурой, используйте команды VM `push argument 0, pop pointer 0`.

Отображение элементов массива

Высокоуровневая ссылка `arr[expression]` компилируется посредством присваивания значения `(arr + expression)` указателю `pointer 1` и обращения к `that 0`.

Отображение констант

- Ссылки на константы `Jack null` и `false` компилируются как `push constant 0`.
- Ссылки на константу `Jack true` компилируются как `push constant 1, neg`. Эта последовательность помещает в стек значение `-1`.
- Ссылки на константу `Jack this` компилируются как `push pointer 0`. Эта команда помещает в стек базовый адрес текущего объекта.

11.3.2. Советы по реализации

В этой главе мы рассмотрели множество концептуальных примеров компиляции. Дадим теперь краткое и формальное изложение всех этих стратегий компиляции.

Обработка идентификаторов

Идентификаторы, используемые для именования переменных, можно обрабатывать с помощью таблиц символов. Любой идентификатор, не найденный во время компиляции корректного кода Jack в таблицах символов, может быть принят за имя подпрограммы или класса. Поскольку правил синтаксиса Jack достаточно для различения этих двух вариантов и поскольку компилятор Jack не выполняет «связывание», необходимости хранить эти идентификаторы в таблице символов нет.

Компиляция выражений

Процедура `compileExpression` должна обрабатывать входные данные как последовательность *term op term op term...* Для этого `compileExpression` должна реализовывать алгоритм `codeWrite` (иллюстрация 11.4), расширенный для обработки всех возможных терминов (*term*), указанных в грамматике Jack (иллюстрация 11.5). Из анализа правил грамматики видно, что большинство действий при компиляции *выражений* предполагают компиляцию лежащих в их основе терминов *terms*. Особенно это верно в свете нашей рекомендации о том, чтобы компиляция вызовов процедур обрабатывалась непосредственно компиляцией *terms* (замечания по реализации API `CompilationEngine`, раздел 10.3).

Грамматика *выражений* (*expressions*) и, следовательно, соответствующая процедура `compileExpression` по своей сути рекурсивны. Например, когда процедура `compileExpression` обнаруживает левую скобку, она должна рекурсивно вызвать `compileExpression` для обработки внутреннего выражения. Такой рекурсивный спуск гарантирует, что внутреннее выражение будет вычислено первым.

За исключением этого правила приоритета, язык Jack не поддерживает *приоритет операторов*. Конечно, можно реализовать и обработку приоритета операторов, но в рамках практического курса «От Nand до “Тетриса”» мы рассматриваем ее как возможное дополнительное расширение, специфичное для компилятора, а не как стандартную возможность языка Jack.

Выражение $x * y$ компилируется как `push x, push y, call Math.multiply 2`. Выражение x / y компилируется как `push x, push y, call Math.divide 2`. Класс `Math` является частью ОС и задокументирован в приложении 6. Этот класс будет разработан в главе 12.

Компиляция строк

Каждая строковая константа «*sss...c*» обрабатывается посредством: 1) переноса длины строки в стек и вызова конструктора `String.new` и 2) переноса кода символа *c* в стек и вызова метода `String appendChar`, по одному разу для каждого символа *c* в строке (набор символов Jack документирован в приложении 5). Как описано в API класса `String` в приложении 6, и конструктор `new`, и метод `appendChar` возвращают строку как возвращаемое значение (то есть помещают объект строки в стек). Это упрощает компиляцию, так как избавляет от необходимости заново помещать строку в стек при каждом вызове `appendChar`.

Компиляция вызовов функций и конструкторов

Компилируемая версия вызова функции или конструктора с *n* аргументами должна: 1) вызвать процедуру `compileExpressionList`, которая вызовет процедуру `compileExpression` *n* раз, и 2) сделать вызов с сообщением о том, что перед вызовом в стек были помещены *n* аргументов.

Компиляция вызовов методов

Компилированная версия вызова метода с *n* аргументами должна: 1) поместить ссылку на объект, для работы над которым вызывается

метод, 2) вызвать процедуру `compileExpressionList`, которая вызовет `compileExpression` n раз, и 3) выполнить вызов с сообщением, что перед вызовом в стек было помещено $n + 1$ аргументов.

Компиляция высказываний `do`

Мы рекомендуем компилировать высказывания `do subroutineCall` так, как если бы они были высказыванием `do expression`, а затем вынимать из стека верхнее значение с помощью команды `pop temp 0`.

Компиляция классов

Приступая к компиляции класса, компилятор создает таблицу символов уровня класса и добавляет в нее все *полевые* и *статические* переменные, объявленные в объявлении класса. Компилятор также создает пустую таблицу символов уровня процедуры. Никакого кода при этом не генерируется.

Компиляция процедур

- Приступая к компиляции процедуры (*конструктора, функции или метода*), компилятор инициализирует таблицу символов процедуры. Если процедура — это *метод*, компилятор добавляет в таблицу символов привязку `<this, className, arg, 0>`.
- Далее компилятор добавляет в таблицу символов все объявленные в списке параметров процедуры параметры, если таковые имеются. Далее обрабатывает все объявления `var`, если таковые имеются, добавляя в таблицу символов все локальные переменные процедуры.
- На этом этапе компилятор начинает генерировать код, стартуя с команды `function className.subroutineName nVars`, где $nVars$ — количество локальных переменных в процедуре.
- Если процедура является *методом*, компилятор генерирует код `push argument 0, pop pointer 0`. Эта последовательность связывает сегмент виртуальной памяти `this` с базовым адресом объекта, для которого был вызван метод.

Компиляция конструкторов

- Сначала компилятор выполняет все действия, описанные в предыдущем разделе, заканчивая генерацией команды `function className.constructorName nVars`.
- Далее компилятор генерирует код `push constant nFields, call Memory.alloc 1, pop pointer 0`, где *nFields* — количество полей в скомпилированном классе. Это приводит к выделению блока памяти из *nFields* 16-битных слов и привязке сегмента виртуальной памяти `this` к базовому адресу вновь выделенного блока.
- Скомпилированный конструктор должен заканчиваться командами `push pointer 0, return`. Эта последовательность возвращает вызывающей стороне базовый адрес созданного конструктором нового объекта.

Компилирование методов `void` и функций `void`

Предполагается, что каждая функция ВМ перед возвратом помещает в стек некое значение. При компиляции `void`-метода Jack или `void`-функции Jack принято завершать сгенерированный код командами `push constant 0, return`.

Компиляция массивов

Высказывания вида `let arr[expression1] = expression2` компилируются с помощью описанной в конце раздела 11.1.6 стратегии. *Совет по реализации:* при работе с массивами никогда не бывает необходимости использовать записи `that`, индекс которых больше 0.

Операционная система

Рассмотрим высокоуровневое выражение `Math.sqrt((dx * dx) + (dy * dy))`. Компилятор компилирует его в команды ВМ `push dx, push dx, call Math.multiply 2, push dy, push dy, call Math.multiply 2, add, call Math.sqrt 1`, где *dx* и *dy* — отображения

dx и dy в таблице символов. Этот пример иллюстрирует два способа, с помощью которых во время компиляции в игру вступают службы операционной системы. Во-первых, некоторые высокоуровневые абстракции, такие как выражение $x * y$, компилируются посредством генерации кода, вызывающего процедуры ОС, например, `Math.multiply`. Во-вторых, если выражение Jack включает в себя высокоуровневый вызов процедуры ОС, например `Math.sqrt(x)`, компилятор генерирует код ВМ, который делает точно такой же вызов с использованием постфиксного синтаксиса ВМ.

ОС включает в себя восемь классов, документированных в приложении 6. В рамках практического курса «От Nand до “Тетриса”» представлены две различные реализации этой ОС — *нативная* и *эмулированная*.

Нативная реализация ОС

В проекте 12 вы разработаете библиотеку классов ОС на языке Jack и скомпилируете ее с помощью компилятора Jack. В результате компиляции будет получено восемь `.vm`-файлов, содержащих нативную реализацию ОС. Если вы поместите эти восемь `.vm`-файлов в ту же папку, где хранятся `.vm`-файлы, полученные в результате компиляции *любой* программы Jack, для скомпилированного кода ВМ станут доступны все функции ОС, поскольку они принадлежат к одной и той же кодовой базе.

Эмулированная реализация ОС

Поставляемый эмулятор ВМ представляет собой программу, написанную на языке Java и содержащую основанную на Java реализацию ОС Jack. Всякий раз, когда загруженный в эмулятор код ВМ вызывает функцию ОС, эмулятор проверяет, существует ли функция ВМ с таким именем в загруженной базе кода. Если да, то он выполняет функцию ВМ. В противном случае вызывает встроенную реализацию этой функции ОС. В итоге получается следующее: если вы используете поставляемый эмулятор ВМ для выполнения кода ВМ, сгенерированного вашим компилятором, как это мы делаем в проекте 11, то вам

не нужно беспокоиться о конфигурации ОС; эмулятор будет самостоятельно обслуживать все вызовы ОС.

11.3.3. Архитектура программного обеспечения

Предлагаемая архитектура компилятора основана на синтаксическом анализаторе, описанном в главе 10. В частности, мы предлагаем постепенно превращать синтаксический анализатор в полноценный компилятор с использованием следующих модулей.

- `JackCompiler`: основная программа, устанавливающая и вызывающая другие модули.
- `JackTokenizer`: токенизатор для языка Jack.
- `SymbolTable`: ведет учет всех переменных, встречающихся в коде Jack.
- `VMWriter`: пишет код ВМ.
- `CompilationEngine`: механизм нисходящей рекурсивной компиляции.

JackCompiler

Этот модуль управляет процессом компиляции. Он работает либо с именем файла вида `Xxx.jack`, либо с именем папки, содержащей один или несколько таких файлов. Для каждого исходного файла `Xxx.jack` программа:

- 1) создает `JackTokenizer` из входного файла `Xxx.jack`;
- 2) создает выходной файл `Xxx.vm`;
- 3) использует `CompilationEngine`, `SymbolTable` и `VMWriter` для разбора входного файла и вывода транслированного кода ВМ в выходной файл.

Мы не предоставляем API для этого модуля, предлагая вам реализовать его по своему усмотрению. Помните, что при компиляции файла `.jack` первой процедурой должна вызываться `compileClass`.

JackTokenizer

Этот модуль идентичен токенизатору, построенному в проекте 10. Смотрите API в разделе 10.3.

SymbolTable

Этот модуль предоставляет сервисы по созданию, заполнению и использованию таблиц символов, в которых хранятся *имя*, *тип*, *вид* и текущий *индекс* для каждого вида символов. Пример показан на иллюстрации 11.2.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	—	—	Создает новую таблицу символов
reset	—	—	Опустошает таблицу символов и обнуляет четыре индекса. Должна вызываться при начале компиляции объявления процедуры
define	name (string) type (string) kind (STATIC, FIELD, ARG или VAR)	—	Определяет (добавляет в таблицу) новую переменную с заданным именем (name), типом (type) и видом (kind). Присваивает ей индексное значение этого вида и добавляет 1 к индексу
varCount	kind (STATIC, int FIELDS, ARG или VAR)		Возвращает количество переменных данного вида (kind), уже определенных в таблице
kindOf	name (sting)	(STATIC, FIELD, ARG, VAR, NONE)	Возвращает вид (kind) названного идентификатора. Если идентификатор не найден, возвращает NONE
typeOf	name (sting)	string	Возвращает тип (type) названной переменной
indexOf	name (sting)	int	Возвращает индекс (index) названной переменной

Примечание по реализации: во время компиляции файла класса Jack компилятор использует два экземпляра SymbolTable.

VMWriter

Этот модуль содержит набор простых процедур для записи в выходной файл команд VM.

<i>Процедура</i>	<i>Аргументы</i>	<i>Возвращает</i>	<i>Функция</i>
Конструктор/ инициализатор	Выходной файл / поток	—	Создает новый файл .vm/ поток и подготавливает его к записи.
writePush	segment (CONSTANT, ARGUMENT, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) index (int)	—	Записывает команду VM push
writePop	segment (AGRUMENT, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) index (int)	—	Записывает команду VM pop
writeArithmetic	command (ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT)	—	Записывает арифметико- логическую команду VM
writeLabel	label (string)	—	Записывает команду VM label
writeGoto	label (string)	—	Записывает команду VM goto
writeIf	label (string)	—	Записывает команду VM if-goto
writeCall	label (string)	—	Записывает команду VM call
writeFunction	label (string)	—	Записывает команду VM function
writeReturn	—	—	Записывает команду VM return
close	—	—	Закрывает файл вывода / поток

CompilationEngine

Этот модуль запускает процесс компиляции. Хотя API `CompilationEngine` практически идентичен API, представленному в главе 10, мы повторяем его здесь для удобства.

Модуль `CompilationEngine` получает входные данные от модуля `JackTokenizer` и использует модуль `VMWriter` для записи выходного кода ВМ (вместо создаваемого в проекте 10 XML). Выходные данные генерируются серией процедур `compilexxx`, каждая из которых предназначена для компиляции определенной конструкции языка Jack `xxx` (например, `compileWhile` генерирует код ВМ, реализующий высказывание `while`). Порядок действий между этими процедурами следующий: каждая процедура `compilexxx` получает на вход и обрабатывает все составляющие `xxx` токены, продвигает токенизатор точно за эти токены и выдает на выход код ВМ, реализующий семантику `xxx`. Если `xxx` — это часть выражения и, следовательно, имеет значение, то выдаваемый код ВМ должен вычислить это значение и поместить его на вершину стека. Как правило, каждая процедура `compilexxx` вызывается только в том случае, если текущий токен равен `xxx`. Поскольку первым токеном в допустимом файле `.jack` должно быть ключевое слово `class`, процесс компиляции начинается с вызова процедуры `compileClass`.

Процедура	Аргументы	Возвращает	Функция
Конструктор/ инициализатор	Входной файл / поток Выходной файл / поток	—	Создает новый модуль компилятора с заданными входом и выходом. Следующая вызываемая процедура — <code>compileClass</code>
<code>compileClass</code>	—	—	Компилирует полностью класс
<code>compileClassVarDec</code>	—	—	Компилирует объявление статической или полевой переменной
<code>compileSubroutine</code>	—	—	Компилирует полностью метод, функцию или конструктор

<code>compileParameterList</code>	—	—	Компилирует (возможно, пустой) список параметров. Не обрабатывает токены закрывающих скобок (и)
<code>compileSubroutineBody</code>	—	—	Компилирует тело процедуры
<code>compileVarDec</code>	—	—	Компилирует объявление <code>var</code>
<code>compileStatements</code>	—	—	Компилирует последовательность высказываний. Не обрабатывает токены закрывающих фигурных скобок { и }
<code>compileLet</code>	—	—	Компилирует высказывание <code>let</code>
<code>compileIf</code>	—	—	Компилирует высказывание <code>if</code> , возможно, с последующим <code>else</code>
<code>compileWhile</code>	—	—	Компилирует высказывание <code>while</code>
<code>compileDo</code>	—	—	Компилирует высказывание <code>do</code>
<code>compileReturn</code>	—	—	Компилирует высказывание <code>return</code>
<code>compileExpression</code>	—	—	Компилирует выражение
<code>compileTerm</code>	—	—	Компилирует термин <i>term</i> . Если текущий токен — <i>идентификатор</i> , процедура должна определить его как <i>переменную, элемент массива или вызов процедуры</i> . Для определения достаточно одного прогностического токена [, (или . . Любой другой токен — не часть этого термина, и обрабатывать его не следует
<code>compileExpressionList</code>	—	<code>int</code>	Компилирует (возможно, пустой) список выражений, разделенных запятой. Возвращает количество выражений в списке

Примечание: следующие правила грамматики Jack не имеют соответствующих процедур `compilexxx` в модуле `CompilationEngine`: `type`, `className`, `subroutineName`, `varName`, `statement`, `subroutineCall`.

Логика разбора этих правил должна обрабатываться процедурами, реализующими правила, которые на них ссылаются. Грамматика языка Jack изложена в разделе 10.2.1.

Прогнозирующий просмотр токенов: необходимость в прогнозирующем просмотре токенов и предлагаемое решение для работы с ними обсуждаются в разделе 10.3, сразу после изложения API `CompilationEngine`.

11.4. Проект

Задача: расширить синтаксический анализатор, созданный в главе 10, до полноценного компилятора Jack. Применить свой компилятор ко всем описанным ниже тестовым программам. Выполнить каждую транслированную программу и убедиться в том, что она работает согласно документации.

В этой версии компилятора предполагается, что исходный код Jack не содержит ошибок. Проверку на наличие ошибок и их обработку можно будет добавить в последующие версии ассемблера, но в проект 11 они не входят.

Ресурсы: основной инструмент, который вам понадобится, — это язык программирования, на котором вы будете реализовывать компилятор. Вам также понадобится прилагаемый эмулятор ВМ для тестирования кода ВМ, сгенерированного вашим компилятором. Поскольку компилятор реализуется посредством расширения синтаксического анализатора, построенного в проекте 10, также понадобится исходный код анализатора.

Этапы реализации

Мы предлагаем превратить в окончательный компилятор синтаксический анализатор, построенный в проекте 10. В частности, постепенно заменить процедуры, генерирующие пассивный вывод в виде XML,

на процедуры, генерирующие исполняемый код ВМ. Это можно сделать в два основных этапа разработки.

(Этап 0: создание резервной копии кода синтаксического анализатора, разработанного в проекте 10.)

Этап 1. Таблица символов: начните со сборки модуля `SymbolTable` и используйте его для расширения синтаксического анализатора, созданного в проекте 10. Встречая в исходном коде идентификатор, например `foo`, собранный ранее синтаксический анализатор выводит XML-строку `<identifier> foo </identifier>`. Переделайте синтаксический анализатор так, чтобы выводить следующую информацию о каждом идентификаторе:

- *Имя.*
- *Категория* (`field`, `static`, `var`, `arg`, `class`, `subroutine`).
- *Индекс:* если категория идентификатора — `field`, `static`, `var` или `arg`, то это текущий индекс, присвоенный идентификатору таблицей символов.
- *Использование:* *объявляется* ли идентификатор в данный момент (например, появляется в объявлении переменной `Jack static / field / var`) или *используется* (например, идентификатор появляется в выражении `Jack`).

Сделайте так, чтобы ваш синтаксический анализатор выводил эту информацию как часть XML-разметки, воспользовавшись тегами разметки по вашему выбору.

Протестируйте новый модуль `SymbolTable` и описанную выше новую функциональность, запустив ваш расширенный синтаксический анализатор на тестовых программах `Jack`, поставляемых в проекте 10. Если ваш расширенный синтаксический анализатор правильно выводит описанную выше информацию, это означает, что вы полностью реализовали анализ семантики `Jack`-программ. На этом этапе можно перейти к разработке полнофункционального компилятора и генерации кода ВМ вместо XML. Это можно сделать постепенно, как описано далее.

(Этап 1,5: сделайте резервную копию расширенного синтаксического анализатора.)

Этап 2. Генерация кода: мы предоставляем шесть прикладных программ, предназначенных для постепенного модульного тестирования генерации кода вашего компилятора Jack. Рекомендуем разрабатывать и тестировать развивающийся компилятор на тестовых программах в указанном порядке. Таким образом вы будете наращивать возможности компилятора разумными темпами и в соответствии с требованиями, предъявляемыми каждой тестовой программой.

Обычно, компилируя программу и сталкиваясь с проблемами, приходят к выводу, что программа испорчена. В данном проекте ситуация прямо противоположная. Все предоставленные тестовые программы не содержат ошибок. Поэтому если их компиляция приведет к каким-либо ошибкам, то исправлять придется компилятор, а не программы. В частности, для каждой тестовой программы мы рекомендуем придерживаться следующей последовательности.

1. Скомпилируйте папку с программой, используя разрабатываемый вами компилятор. В результате у вас должен получиться один файл `.vm` для каждого исходного файла `.jack` в данной папке.
2. Просмотрите сгенерированные файлы VM. При наличии заметных проблем исправьте свой компилятор и перейдите к шагу 1. Помните: все поставляемые тестовые программы не содержат ошибок.
3. Загрузите папку с программой в эмулятор VM и запустите загруженный код. Обратите внимание, что каждая из шести поставляемых тестовых программ содержит конкретные рекомендации по выполнению; проверьте скомпилированную программу (транслированный код VM) в соответствии с этими рекомендациями.
4. Если программа ведет себя неправильно или эмулятор виртуальной машины выдает сообщение об ошибке, исправьте компилятор и перейдите к шагу 1.

Тестирование программ

Seven: проверяет, как компилятор обрабатывает простую программу, содержащую арифметическое выражение с целочисленными константами, высказывание `do` и высказывание `return`. В частности, программа вычисляет выражение $1 + (2 * 3)$ и печатает его значение в левом верхнем углу экрана. Для проверки правильности трансляции запустите транслированный код в эмуляторе ВМ и проверьте, правильно ли он отображает 7.

ConvertToBin: проверяет, как компилятор обрабатывает все процедурные элементы языка Jack: выражения (без массивов или вызовов методов), функции и операторы `if`, `while`, `do`, `let` и `return`. Программа не проверяет обработку методов, конструкторов, массивов, строк, статических переменных и переменных поля. В частности, программа получает 16-битное десятичное значение из `RAM[8000]`, преобразует его в двоичное и сохраняет отдельные биты в `RAM[8001...8016]` (каждый участок памяти будет содержать 0 или 1). Перед началом преобразования программа инициализирует `RAM[8001...8016]` значением `-1` (записывает его в эти участки памяти). Для проверки правильности трансляции загрузите транслированный код в эмулятор ВМ и выполните следующие действия.

- Поместите (интерактивно, через графический интерфейс эмулятора) в `RAM[8000]` некоторое десятичное значение.
- Запустите программу на несколько секунд, затем остановите ее.
- Проверьте (визуально), что ячейки памяти `RAM[8001...8016]` содержат правильные биты и что ни одна из них не содержит `-1`.

Square: проверяет, как компилятор обрабатывает объектно-ориентированные особенности языка Jack: конструкторы, методы, поля и выражения, включающие вызовы методов. Не проверяет работу со статическими переменными. В частности, эта программа из нескольких классов представляет собой простую интерактивную игру, позволяющую перемещать черный квадрат по экрану с помощью четырех клавиш со стрелками на клавиатуре.

Во время перемещения размер квадрата можно увеличивать и уменьшать, нажимая соответственно клавиши *z* и *x*. Чтобы выйти из игры, нажмите клавишу *q*. Для проверки правильности трансляции запустите транслированный вашим компилятором код в эмуляторе ВМ и убедитесь, что игра работает так, как ожидается.

Average: проверяет, как компилятор обрабатывает массивы и строки. Это делается посредством вычисления среднего значения целых чисел заданной пользователем последовательности. Для проверки правильности трансляции запустите транслированный код в эмуляторе ВМ и следуйте отображаемым на экране инструкциям.

Pong: полный тест обработки компилятором объектно-ориентированного приложения, включая работу с объектами и статическими переменными. В классической игре Pong шарик движется случайным образом, отскакивая от краев экрана. Пользователь пытается отбить мяч маленькой ракеткой, которую можно перемещать, нажимая клавиши со стрелками влево и вправо. Каждый раз, когда ракетка отбивает мяч, пользователь зарабатывает очко, а ракетка немного уменьшается, что усложняет игру. При промахе мяч падает за пределы экрана, и игра заканчивается. Для проверки правильности трансляции запустите транслированный код в эмуляторе виртуальной машины и сыграйте в игру. Обязательно наберите несколько очков, чтобы проверить часть программы, выводящую на экран счет.

ComplexArrays: проверяет, как компилятор обрабатывает сложные ссылки на массивы и выражения. Для этого программа выполняет пять сложных вычислений с использованием массивов. Для каждого такого вычисления она выводит на экран ожидаемый результат, а также результат, вычисленный скомпилированной программой. Для проверки правильности трансляции запустите транслированный код в эмуляторе виртуальной машины и убедитесь, что ожидаемые и фактические результаты совпадают.

Веб-версия проекта 11 доступна на сайте www.nand2tetris.org.

11.5. Перспектива

Jack — объектно-ориентированный язык программирования общего назначения. По замыслу разработчиков, он должен быть относительно простым языком. Его простота позволила нам обойти несколько сложных вопросов компиляции. Например, несмотря на то что Jack выглядит как типизированный язык, это не так: все типы данных Jack (`int`, `char` и `Boolean`) имеют ширину 16 бит, что позволяет компиляторам Jack игнорировать почти всю информацию о типах. В частности, при компиляции и оценке выражений компилятором Jack не нужно определять их типы. Единственное исключение — компиляция вызовов методов вида `x.m()`, что требует определения типа класса `x`. Еще одно проявление простоты типов в Jack — отсутствие типизации элементов массивов.

В отличие от Jack, большинство языков программирования имеют богатые системы типов, предъявляющие компиляторам дополнительные требования: под разные типы переменных необходимо выделять разное количество памяти; преобразование из одного типа в другой требует неявных и явных операций приведения; компиляция простого выражения типа `x + y` сильно зависит от типов `x` и `y`, и т. д.

Еще одно существенное упрощение — это то, что язык Jack не поддерживает наследования. В языках, поддерживающих наследование, обработка вызовов методов типа `x.m()` зависит от принадлежности объекта `x` к классу, что можно определить только во время выполнения. Поэтому компиляторы объектно-ориентированных языков, поддерживающих наследование, должны рассматривать все методы как виртуальные и определять их принадлежность к классу в соответствии с типом объекта, к которому во время выполнения программы применяется метод. Поскольку Jack не поддерживает наследования, все вызовы методов можно компилировать статически во время компиляции.

Еще одна общая черта объектно-ориентированных языков, не поддерживаемая Jack, — различие между приватными и публичными членами класса. В Jack все статические и полевые переменные являются

приватными (распознаются только в пределах класса, в котором они объявлены), а все процедуры — публичными (могут вызываться из любого класса).

Отсутствие реальной типизации, наследования и публичных полей позволяет компилировать классы независимо, без доступа к коду любого другого класса. К полям других классов никогда не обращаются напрямую, а все ссылки на методы других классов являются «поздними» и осуществляются только по имени.

Многие упрощения языка Jack не существенны, и их можно устранить без особых усилий. Например, можно легко расширить язык с помощью операторов `for` и `switch`. Аналогично можно добавить возможность присваивать переменным типа `char` символьные константы типа `'c'`, что в настоящее время не поддерживается языком.

Наконец, наши стратегии генерации кода не уделяли никакого внимания оптимизации. Рассмотрим высказывание на типичном языке высокого уровня: C++. Базовый компилятор транслирует его в серию низкоуровневых операций ВМ: `push c, push 1, add, pop c`. Далее транслятор ВМ преобразует каждую из этих команд ВМ в несколько инструкций машинного уровня, в результате чего получается довольно объемный кусок кода. В то же время оптимизированный компилятор заметит, что мы имеем дело с простым приращением (инкрементом), и транслирует его, допустим, в две машинные инструкции `@c` и `M = M + 1`. Конечно, это лишь один из примеров того, что ожидается от промышленно мощных компиляторов. В целом авторы компиляторов тратят много усилий и изобретательности, чтобы при создании кода обеспечить экономию времени и пространства.

В практическом курсе «От Nand до “Тетриса”» внимание эффективности уделяется редко, за одним большим исключением в виде операционной системы. ОС Jack основывается на эффективных алгоритмах и оптимизированных структурах данных, о чем мы подробно расскажем в следующей главе.

12. Операционная система

Цивилизация идет по пути прогресса, расширяя количество операций, которые мы можем выполнять, не задумываясь о них.

— Альфред Норт Уайтхед, «Введение в математику» (1911)

В главах 1–6 мы описали и построили аппаратную архитектуру общего назначения. В главах 7–11 разработали иерархию программного обеспечения, которая делает аппаратуру пригодной для использования, кульминацией чего стало создание современного объектно-ориентированного языка. Поверх аппаратной платформы можно разработать и реализовать и другие языки программирования, для каждого из которых потребуется свой компилятор.

Последний важный элемент, которого не хватает для составления полной картины, — *операционная система* (ОС). ОС предназначена для устранения разрывов между аппаратным и программным обеспечением компьютера и для того, чтобы сделать компьютерную систему более доступной для программистов, компиляторов и пользователей. Например, чтобы вывести на экран текст `Hello World`, необходимо включить и выключить несколько сотен пикселей в определенных местах экрана. Это можно сделать, обратившись к спецификации аппаратного обеспечения и написав код, включающий и выключающий биты в выбранных местах оперативной памяти. Очевидно, что программисты высокого уровня ожидают интерфейса получше. Они хотят просто написать высказывание `print ("Hello World")`, а о деталях пусть беспокоится кто-то другой. Вот тут-то и вступает в дело операционная система.

Во всей этой главе термин «*операционная система*» используется в довольно широком смысле. Наша ОС минимальна и нацелена на 1) инкапсуляцию низкоуровневых служб, специфичных для аппаратного обеспечения, в высокоуровневые программные сервисы, удобные для программистов, и 2) расширение языков высокого уровня с помощью часто используемых функций и абстрактных типов данных. В этом смысле различие между собственно *операционной системой* и *стандартной библиотекой классов* довольно размытое. Современные языки программирования включают в себя многие стандартные сервисы операционной системы, такие как управление графикой, управление памятью, многозадачность и множество других расширений, образующих так называемую *стандартную библиотеку классов* языка. Следуя этой модели, ОС Jack упакована как набор вспомогательных классов, каждый из которых предоставляет соответствующие сервисы через вызовы процедур Jack. Полный API ОС изложен в приложении 6.

Программисты высокого уровня ожидают от ОС предоставления сервисов через хорошо продуманные интерфейсы, скрывающие от их прикладных программ подробности управления аппаратным обеспечением. Для этого код ОС должен работать как можно ближе к аппаратному обеспечению, практически напрямую манипулируя памятью, вводом/выводом и устройствами обработки. Кроме того, поскольку ОС поддерживает выполнение каждой программы, работающей на компьютере, она должна быть высокоэффективной. Например, прикладные программы постоянно создают и уничтожают объекты и массивы. Поэтому лучше делать это быстро и экономно. Любой выигрыш во времени и объеме данных существенно влияет на производительность всех прикладных программ, зависящих от работы ОС.

Операционные системы обычно пишутся на языке высокого уровня и компилируются в двоичную форму. Наша ОС не исключение — она написана на языке Jack, подобно тому, как Unix был написан на языке C. Как и язык C, Jack разрабатывался с ориентацией на достаточно «низкий уровень», позволяющий при необходимости обеспечить тесную связь с аппаратным обеспечением.

Глава начинается с относительно длинного раздела «Основы», в котором описаны ключевые алгоритмы, обычно используемые в реализации ОС. К ним относятся математические операции, работа со строками, управление памятью, вывод текста и графики, а также ввод с клавиатуры. За этим алгоритмическим введением следует раздел «Спецификация», описывающий ОС Jack, и раздел «Реализация», предлагающий руководство по созданию ОС с использованием описанных алгоритмов. Как обычно, раздел «Проект» содержит необходимые указания и материалы для поэтапного создания и модульного тестирования всей ОС.

В главе отражены ключевые элементы системно-ориентированной программной инженерии и информатики. С одной стороны, мы описываем приемы программирования для разработки низкоуровневых системных сервисов, а также приемы «широкомасштабного» программирования для интеграции и оптимизации сервисов ОС. С другой стороны, рассказываем про набор элегантных и высокоэффективных алгоритмов, каждый из которых можно назвать настоящей жемчужиной информатики.

12.1. Основы

Компьютеры обычно подключаются к различным устройствам ввода/вывода, таким как клавиатура, экран, мышь, накопитель, сетевая карта, микрофон, динамики и т. д. Каждое из этих устройств ввода/вывода имеет свои собственные электромеханические особенности, поэтому чтение и запись данных на них сопряжены с множеством технических подробностей. Языки высокого уровня абстрагируются от этого, предлагая высокоуровневые абстракции типа `let n = Keyboard.readInt("Введите число: ")`. Давайте разберемся, что нужно сделать, чтобы реализовать эту, казалось бы, простую операцию ввода данных.

Сначала мы привлекаем внимание пользователя, выводя на экран запрос `Введите число: .` Для этого нужно создать объект типа `String` и инициализировать его массивом символов `'В', ' ', 'е'`

и т. д. Затем мы должны вывести эту строку на экран, по одному символу за раз, одновременно обновляя позицию *курсора* для отслеживания физического положения следующего символа. После отображения подсказки Введите число: нужно запустить цикл, ожидающий, пока пользователь не нажмет на клавиатуре какие-либо клавиши, предположительно обозначающие цифры. Для этого необходимо знать, как: 1) перехватывать нажатие клавиши, 2) получать один введенный символ, 3) добавлять символ к строке и 4) преобразовывать строку в целочисленное значение.

Если это описание показалось вам сложным, то учтите, что мы еще многое оставили за кадром. Например, что именно подразумевается под «созданием строкового объекта», «отображением символа на экране» и «получением на входе нескольких символов»?

Начнем с «создания строкового объекта». Строковые объекты не появляются просто так из воздуха уже полностью сформированные. Каждый раз, желая создать объект, мы должны найти свободное место для представления объекта в оперативной памяти, пометить эту область памяти как используемую и не забыть освободить ее, когда объект больше не нужен. Переходя к абстракции «отобразить символ», отметим, что, строго говоря, никакие «символы на экране не отображаются». Единственное, что можно отобразить физически, — это отдельные пиксели. Таким образом, мы должны разработать и определить *шрифт* символа, вычислить, где в карте памяти экрана находятся биты, представляющие изображение данного шрифта, а затем включать и выключать эти биты по мере необходимости. Наконец, чтобы «получить на входе несколько символов», мы должны организовать цикл, который будет не только прислушиваться к клавиатуре и накапливать символы по мере их ввода, но и позволять пользователю при ошибке стирать эти символы и перепечатывать их, не говоря уже о необходимости отражать каждое из данных действий на экране для поддержки визуальной обратной связи.

Забота о такой сложной закадровой работе и входит в задачи операционной системы. Выполнение выражения `let n = Keyboard.readInt("Введите число: ")` влечет за собой множество вызовов функций ОС, связанных с такими различными вопросами, как

распределение памяти, управление вводом, управление выводом и обработка строк. Компиляторы пользуются сервисами ОС абстрактно, внедряя вызовы функций ОС в компилируемый код, как мы видели в предыдущей главе. В этой главе мы исследуем, *как эти функции реализуются на практике*. Конечно, рассмотренные выше примеры — лишь небольшое подмножество обязанностей ОС. Например, мы не упомянули математические операции, графический вывод и другие часто востребованные сервисы. Хорошая новость заключается в том, что хорошо написанная ОС может интегрировать эти разнообразные и, казалось бы, несвязанные между собой задачи элегантно и эффективным способом с помощью потрясающих алгоритмов и структур данных. Именно им и посвящена данная глава.

12.1.1. Математические операции

В основе почти любой компьютерной программы лежат четыре арифметические операции: *сложение, вычитание, умножение и деление*. Если выражения, использующие некоторые из этих операций, содержатся в цикле, выполняемом миллионы раз, то, естественно, лучше реализовать их как можно более эффективно. Обычно сложение реализуется аппаратно, на уровне АЛУ, а вычитание можно легко реализовать благодаря дополнительному коду, или методу «дополнения до двух». Другие арифметические операции можно реализовывать либо аппаратно, либо программно, в зависимости от соображений стоимости/производительности. Перейдем теперь к описанию эффективных алгоритмов для умножения, деления и вычисления квадратных корней. Такие алгоритмы также допускают как программные, так и аппаратные реализации.

Эффективность прежде всего

Математические алгоритмы создаются для работы с n -битными значениями, где n — это обычно 16, 32 или 64 бита, в зависимости от типов данных операндов. Как правило, мы ищем алгоритмы, время работы

которых является многочленной функцией от размера слова n . Алгоритмы, время работы которых зависит от значений n -битных чисел, неприемлемы, поскольку они зависят от n экспоненциально. Предположим, например, что мы реализуем операцию умножения $x \times y$ «по-простому», используя алгоритм повторного сложения для $i = 1 \dots y$ {sum = sum + x}. Если число y 64-битное, его значение может быть гораздо больше 9 000 000 000 000 000 000, и в таком случае описанный цикл до окончательной своей остановки будет выполняться миллиарды лет.

В отличие от них, время работы описанных ниже алгоритмов умножения, деления и вычисления квадратного корня зависит не от значений n -битных чисел, которые могут достигать 2^n , а от n , то есть количества их битов. И это лучшее, на что мы можем надеяться, когда речь заходит об эффективности арифметических операций.

Для обозначения времени работы «порядка величины n » мы будем пользоваться выражением $O(n)$ (так называемое « O большое»). Время работы всех арифметических алгоритмов, описанных в этой главе, — $O(n)$, где n — ширина битов на входе.

Умножение

Рассмотрим стандартный метод умножения, которому учат в начальной школе. Чтобы вычислить произведение 356 и 73, мы записываем два числа одно над другим с выравниванием по правой стороне. Затем умножаем 356 на 3. Далее сдвигаем число 356 на одну позицию влево и умножаем 3560 на 7 (это то же самое, что умножить 356 на 70). Наконец, мы складываем столбцы этих двух операций умножения и получаем результат. В основе процедуры лежит тот факт, что $356 \times 73 = 356 \times 70 + 356 \times 3$. Двоичная версия этой процедуры показана на иллюстрации 12.1 на другом примере.

Примечание: алгоритмы, представленные в этой главе, написаны на понятном синтаксисе псевдокода. Мы используем отступы для выделения блоков кода, что избавляет от необходимости использовать

фигурные скобки или ключевые слова «begin/end». Например, на иллюстрации 12.1 $sum = sum + shiftedx$ относится к одночастному высказыванию `if`, а $shiftedx = 2 * shiftedx$ представляет собой вторую часть двухчастного высказывания `for`.

Рассмотрим процедуру умножения, показанную на иллюстрации 12.1 слева. Для каждого i -го бита y мы сдвигаем x i раз влево (то же самое, что умножить x на 2^i). Затем смотрим на i -й бит y : если он равен 1, мы добавляем сдвинутый x к накапливаемому значению суммы; в противном случае ничего не делаем. Алгоритм, показанный справа, формализует эту процедуру. Обратите внимание, что $2 * shiftedx$ можно эффективно вычислить либо сдвигом влево побитового представления $shiftedx$, либо прибавлением $shiftedx$ к самому себе. Любая из этих операций легко поддается примитивным аппаратным операциям.

$$\begin{array}{r}
 x = 27 = \dots 000011011 \\
 y = 9 = \dots 000001001 \quad i\text{-й бит } y \\
 \hline
 \dots 000011011 \quad 1 \\
 \dots 000110110 \quad 0 \\
 \dots 001101100 \quad 0 \\
 \dots 011011000 \quad 1 \\
 \hline
 x * y = 243 = \dots 011110011 \quad \text{sum}
 \end{array}$$

```

// Возвращает  $x * y$ , где  $x, y \geq 0$ .
multiply(x, y):
    sum = 0
    shiftedx = x
    for i = 0 ... n - 1 do
        if ((i-й бит y) == 1)
            sum = sum + shiftedx
            shiftedx = 2 * shiftedx
    return sum

```

Иллюстрация 12.1. Алгоритм умножения.

Время работы: алгоритм умножения выполняет n итераций, где n — битовая ширина входного значения y . На каждой итерации алгоритм выполняет несколько операций сложения и сравнения. Из этого следует, что общее время работы алгоритма равно $a + b \cdot n$, где a — время инициализации нескольких переменных, а b — время, необходимое для выполнения нескольких операций сложения и сравнения. Формально время работы алгоритма равно $O(n)$, где n — ширина битов на входах.

Повторим, что время работы этого алгоритма $x \times y$ не зависит от входных значений x и y ; оно скорее зависит от битовой ширины

входных данных. В компьютерах битовая ширина обычно бывает не-большой фиксированной константой, такой как 16 (`short`), 32 (`int`) или 64 (`long`), в зависимости от типов данных на входе. В платформе Nask битовая ширина для всех типов данных равна 16. Если предположить, что каждая итерация алгоритма умножения состоит примерно из десяти машинных инструкций Nask, то отсюда следует, что каждая операция умножения потребует не более 160 тактовых циклов независимо от размера входов. В отличие от этого, алгоритмы, время работы которых пропорционально не битовой ширине, а значениям входных данных, потребуют $10 \cdot 2^{16} = 655\,360$ тактовых циклов.

Деление

«Самый простой» способ деления двух n -битных чисел x / y заключается в подсчете того, сколько раз y можно вычесть из x , пока остаток не станет меньше y . Время работы этого алгоритма пропорционально значению делимого x и, следовательно, экспоненциально числу битов n , что неприемлемо.

Чтобы ускорить процесс, можно попытаться на каждой итерации вычитать большие куски y из x . Предположим, нужно разделить 175 на 3. Для начала зададим вопрос: какое наибольшее число $x = (90, 80, 70, \dots, 20, 10)$ можно подобрать так, чтобы $3 \cdot x \leq 175$? Ответ: 50. Другими словами, нам удалось вычесть из 175 целых пятьдесят троек, сэкономяв пятьдесят итераций по сравнению с «самым простым» подходом. В результате такого ускоренного вычитания у нас получился остаток $175 - 3 \cdot 50 = 25$. Далее спросим, какое самое большое число $x = (9, 8, 7, \dots, 2, 1)$ можно взять, чтобы $3 \cdot x \leq 25$? Ответ: 8, то есть нам удалось еще восемь раз вычесть тройку, так что на этом этапе ответ равен $50 + 8 = 58$. У нас получился остаток $25 - 3 \cdot 8 = 1$, что меньше 3, поэтому мы завершаем процесс и объявляем, что $175 / 3 = 58$ с остатком 1.

Эта техника лежит в основе устрашающей для многих школьников процедуры под названием «деление столбиком». Двоичная версия этого алгоритма идентична, за исключением того, что для ускорения

вычитания мы пользуемся не степенями 10, а степенями 2. Алгоритм выполняет n итераций, где n — количество цифр делимого, и каждая итерация подразумевает несколько операций умножения (фактически сдвига), сравнения и вычитания. И снова у нас получается алгоритм x / y , время работы которого не зависит от значений x и y . Примерное его время работы — $O(n)$, где n — ширина битов на входе.

Записать данный алгоритм, как мы это делали для умножения, несложно. Чтобы было интереснее, на иллюстрации 12.2 показан другой алгоритм деления, столь же эффективный, но более элегантный и простой в реализации.

Предположим, нам нужно разделить 480 на 17. Алгоритм, показанный на рисунке 12.2, основан на рекурсии: $480 / 17 = 2 \cdot (240 / 17) = 2 \cdot (2 \cdot (120 / 17)) = 2 \cdot (2 \cdot (2 \cdot (60 / 17))) = \dots$ и т. д. Глубина этой рекурсии ограничена количеством раз, которым значение y можно умножить на 2, прежде чем оно достигнет x . И заодно это максимальное количество битов, необходимых для представления x . Таким образом, время работы данного алгоритма составляет $O(n)$, где n — битовая ширина чисел на входе.

Небольшая загвоздка в этом алгоритме заключается в том, что каждая операция умножения также требует $O(n)$ операций. Однако при изучении логики алгоритма выясняется, что значение выражения $(2 * q * y)$ можно вычислить без умножения, а вместо этого получить его из его значения на предыдущем уровне рекурсии с помощью сложения.

```
// Возвращает целую часть частного  $x / y$ ,
// где  $x \geq 0$  и  $y > 0$ .
divide( $x, y$ ):
    if ( $y > x$ ) return 0
     $q = \text{divide}(x, 2 * y)$ 
    if  $((x - 2 * q * y) < y)$ 
        return  $2 * q$ 
    else
        return  $2 * q + 1$ 
```

Иллюстрация 12.2. Алгоритм деления.

Квадратный корень

Квадратные корни можно эффективно вычислять различными способами, например, методом Ньютона — Рафсона или разложением в ряд Тейлора. Однако для нашей цели достаточно более простого алгоритма. Функция квадратного корня $y = \sqrt{x}$ обладает двумя привлекательными свойствами. Во-первых, она монотонно возрастающая. Во-вторых, ее обратная функция, $y = x^2$, является по сути умножением, то есть функцией, которую мы уже умеем эффективно вычислять. Отсюда следует, что у нас есть все необходимое для эффективного вычисления квадратных корней методом *двоичного (бинарного) поиска*. Подробности показаны на иллюстрации 12.3.

Поскольку количество итераций в алгоритме двоичного поиска ограничено числом $n / 2$, где n — количество бит в x , время работы алгоритма равно $O(n)$.

Итак, в этом разделе, посвященном математическим операциям, мы описали алгоритмы умножения, деления и вычисления квадратного корня. Примерное время работы каждого из алгоритмов равно $O(n)$, где n — ширина битов входных данных. Также мы отметили, что для компьютеров n — это небольшая константа, например, 16, 32 или 64. Следовательно, каждую операцию сложения, вычитания, умножения и деления можно выполнить быстро, за предсказуемое время, на которое не влияет величина входных данных.

```
// Вычисляет целую часть  $y = \sqrt{x}$ 
// Стратегия: находит такое целое  $y$ , что  $y^2 \leq x < (y + 1)^2$  (для  $0 \leq x < 2^n$ )
// посредством двоичного поиска в диапазоне  $0 \dots 2^{n/2} - 1$ 

sqrt(x):
   $y = 0$ 
  for  $j = (n / 2 - 1) \dots 0$  do
    if  $(y + 2^j)^2 \leq x$  then  $y = y + 2^j$ 
  return  $y$ 
```

Иллюстрация 12.3. Алгоритм вычисления квадратного корня.

12.1.2. Строки

Помимо примитивных типов данных, большинство языков программирования имеют *строковый* тип (*string*), предназначенный для представления последовательностей символов, таких как «Загрузка изображения...» или «ВЫЙТИ». Как правило, абстракция строк обеспечивается классом `String`, частью стандартной библиотеки классов, поддерживающей данный язык. Этот подход также используется в Jack.

Все строковые константы, появляющиеся в программах Jack, реализуются как объекты `String`. Класс `String`, API которого документирован в приложении 6, содержит различные методы обработки строк, такие как добавление символа к строке, удаление последнего символа и т. д. Эти функции, как будет сказано далее в данной главе, реализовать несложно. Сложнее в реализации методы `String`, преобразующие целочисленные значения в строки и строки цифровых символов в целочисленные значения. Рассмотрим алгоритмы, выполняющие эти операции.

Строковое представление чисел: числа внутри компьютеров представлены в виде двоичного кода. Однако люди привыкли иметь дело с числами, записанными в десятичной системе счисления. Поэтому, когда человеку приходится читать или вводить числа, *только тогда* необходимо переводить числа из двоичной системы в десятичную и наоборот. Когда такие числа вводятся с устройства ввода, например с клавиатуры, или выводятся на устройство вывода, например на экран, они представляются в виде строк символов, каждый из которых представляет одну из цифр от 0 до 9. Подмножество соответствующих символов таково.

Символ	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Код символа	48	49	50	51	52	53	54	55	56	57

Полный набор символов Nask приведен в приложении 5. Как видно, цифровые символы можно легко преобразовать в целые числа,

которые они представляют, и наоборот. Целочисленное значение символа c , где $48 \leq c \leq 57$, равно $c - 48$. И наоборот, код символа целого числа x , где $0 \leq x \leq 9$, равен $x + 48$.

Узнав, как работать с однозначными символами, мы можем разработать алгоритмы для преобразования любого целого числа в строку и любой строки цифровых символов в соответствующее целое число. Алгоритмы преобразования могут основываться на итеративной или рекурсивной логике, и на иллюстрации 12.4 представлен один из них.

На основании иллюстрации 12.4 нетрудно сделать вывод, что время работы алгоритмов `int2String` и `string2Int` равно $O(n)$, где n — количество цифровых символов на входе.

int в string:

```
// Возвращает строковое представление
// неотрицательного целого числа.
int2String(val):
    lastDigit = val % 10
    c = символ, представляющий lastDigit
    if (val < 10)
        return c (as a string)
    else
        return int2String(val / 10).appendChar(c)
```

string в int:

```
// Возвращает целое значение строки
// цифровых символов при условии, что str[0] —
// это цифра старшего значащего разряда.
string2Int(str):
    val = 0
    for (i = 0 ... str.length()) do
        d = целочисленное значение str.charAt(i)
        val = val * 10 + d
    return val
```

Иллюстрация 12.4. Преобразование строк в целые числа (`appendChar`, `length` и `charAt` — методы класса `String`).

12.1.3. Управление памятью

Каждый раз, когда программа создает новый массив или новый объект, для их представления должен выделяться блок памяти определенного размера. А когда массив или объект больше не нужны, их пространство в оперативной памяти может быть утилизировано. Эти задачи выполняют две классические функции ОС, называемые `alloc` и `deAlloc`. Данные функции используются компиляторами при генерации низкоуровневого кода для работы с конструкторами и деструкторами, а также программистами высокого уровня по мере необходимости.

Блоки памяти для представления массивов и объектов формируются из выделенной области оперативной памяти, называемой *кучей*. Эта задача также входит в обязанности операционной системы. При своем запуске ОС инициализирует указатель `heapBase`, содержащий базовый адрес кучи в оперативной памяти (в Jack куча начинается сразу после конца стека, при этом `heapBase = 2048`). Мы опишем два алгоритма управления кучей: базовый и улучшенный.

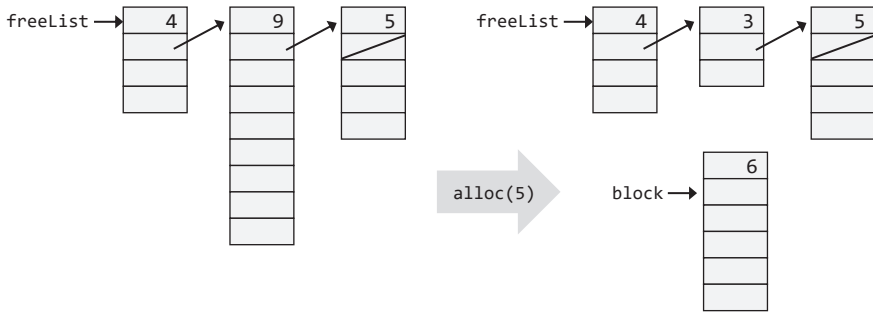
Алгоритм выделения памяти (базовый): структура данных, которой управляет этот алгоритм, представляет собой один указатель с именем `free`, указывающий на начало сегмента кучи, который еще не был выделен. Подробности показаны на иллюстрации 12.5а.

Базовая схема управления кучей явно расточительна, поскольку она никогда не возвращает пространство памяти. Но если ваши прикладные программы используют только несколько небольших объектов и массивов, а также не слишком много строк, можно обойтись и этим алгоритмом.

```
init():  
    free = heapBase  
  
// Выделяет блок памяти размером в size слов.  
alloc(size):  
    block = free  
    free = free + size  
    return block  
  
// Освобождает пространство памяти данного объекта.  
deAlloc(object):  
    do nothing (ничего)
```

Иллюстрация 12.5а. Алгоритм выделения памяти (базовый).

Алгоритм выделения памяти (улучшенный): этот алгоритм управляет связанным списком доступных сегментов памяти, называемым `freeList` (см. иллюстрацию 12.5б). Каждый сегмент в списке начинается с двух полей: *длины* сегмента и указателя на *следующий* сегмент в списке.



```

init():
    freeList = heapBase
    freeList.size = heapSize
    freeList.next = 0

// Выделяет блок памяти размером в size слов.
alloc(size):
    найти freeList с помощью методов best-fit или first-fit
    и получить сегмент размером segment.size ≥ size + 2
    если таковой сегмент не найден, вернуть сообщение об ошибке
    (или попытаться выполнить дефрагментацию)
    block = базовый адрес найденного пространства
    обновить freeList и поля блока
    в соответствии с выделением
    return block

// Освобождает область памяти данного объекта.
deAlloc(object):
    дописать object к концу freeList
  
```

Иллюстрация 12.56. Алгоритм выделения памяти (улучшенный).

Когда алгоритму поручают выделить блок памяти заданного размера, он должен найти в `freeList` подходящий сегмент. Существуют два эвристических метода такого поиска. Метод «лучшего соответствия» (*best-fit*) находит самый короткий сегмент, подходящий для представления требуемого размера, а метод «первого подходящего» (*first-fit*) находит первый достаточно длинный сегмент. Как только подходящий сегмент найден, из него вырезается требуемый блок памяти (участок перед началом возвращаемого блока, `block[-1]`, зарезервирован для хранения его длины, которая будет использоваться при освобождении памяти).

Затем в списке `freeList` обновляется длина данного сегмента, отражая длину той части, которая осталась после выделения. Если в сегменте не осталось памяти или если оставшаяся часть практически слишком мала, сегмент исключается из `freeList`.

При запросе освободить блок памяти неиспользуемого объекта алгоритм добавляет освобождаемый блок в конец `freeList`.

Алгоритмы динамического распределения памяти, подобные показанному на иллюстрации 12.56, могут создавать проблемы фрагментации блоков. Следовательно, необходимо рассмотреть операцию дефрагментации, то есть объединения областей памяти, физически соседних, но логически разделенных на разные сегменты в списке `freeList`. Дефрагментация может выполняться каждый раз при освобождении объекта или когда функция `alloc()` не может найти блок требуемого размера, а также в соответствии с каким-либо периодически возникающим условием.

Peek и Poke: мы завершаем обсуждение управления памятью двумя простыми функциями ОС, не имеющими отношения к распределению ресурсов. `Memory.peek(addr)` возвращает значение оперативной памяти по адресу `addr`, а `Memory.poke(addr, value)` записывает значение слова `value` в оперативную память по адресу `addr`. Эти функции играют роль в различных сервисах ОС, связанных с памятью, включая графические подпрограммы, о которых мы сейчас поговорим.

12.1.4. Графический вывод

Современные компьютеры позволяют осуществлять вывод графических данных, таких как анимация и видео, на цветные экраны с высоким разрешением посредством оптимизированных графических драйверов и специальных графических процессоров (GPU). В практическом курсе «От Nand до “Тетриса”» мы по большей части абстрагировались от такой сложности, сосредоточившись на фундаментальных алгоритмах и техниках отрисовки графики.

Мы исходим из предположения, что компьютер подключен к физическому черно-белому экрану, представленному в виде сетки из строк

и столбцов, на пересечении которых располагаются пиксели. По традиции столбцы нумеруются слева направо, а строки — сверху вниз. Таким образом, пиксель (0,0) расположен в левом верхнем углу экрана.

Также мы исходим из предположения, что экран подключен к компьютерной системе через *карту памяти* — выделенную область оперативной памяти, в которой каждый пиксель представлен одним битом. Экран обновляется в соответствии с этой картой памяти много раз в секунду с помощью внешнего по отношению к компьютеру процесса. Программы, имитирующие работу компьютера, должны эмулировать этот процесс обновления.

Самая основная операция, которую можно выполнить на экране, — это отображение отдельного пикселя, заданного координатами (x, y) . Это достигается посредством включения или выключения соответствующего бита в карте памяти. На основе этой базовой операции строятся другие, такие как рисование линии и рисование окружности. Графический пакет поддерживает *текущий цвет*, который может быть *черным* или *белым*. Все операции рисования используют текущий цвет.

Рисование пикселя (`drawPixel`): отрисовка выбранного пикселя в точке экрана (x, y) осуществляется посредством нахождения соответствующего бита в карте памяти и установки в него значения выбранного цвета. Поскольку оперативная память — это n -битное устройство, такая операция требует чтения и записи n -битного значения. Алгоритм показан на иллюстрации 12.6.

```
//Задать пикселю  $(x, y)$  текущий цвет.
```

```
drawPixel(x, y):
```

```
На основе  $x$  и  $y$  вычислить адрес ОЗУ,  
в котором представлен этот пиксель;
```

```
С помощью Memory.read получить 16-битное  
значение этого адреса;
```

```
С помощью некоей побитовой операции задать  
(только одному) биту значение текущего цвета;
```

```
С помощью Memory.write записать измененное  
16-битное значение обратно по адресу ОЗУ.
```

Иллюстрация 12.6. Рисование пикселя.

Интерфейс карты памяти экрана Hask описан в разделе 5.2.4. Для реализации алгоритма **drawPixel** нужно будет воспользоваться описанной там привязкой экрана к карте памяти.

Рисование линии (drawLine): если мы захотим отобразить непрерывную «линию» между двумя «точками» на сетке из дискретных пикселей, то лучшее, что можно сделать, — это нарисовать серию пикселей вдоль воображаемой линии, соединяющей две точки. «Перо», используемое для рисования линии, может двигаться только в четырех направлениях: вверх, вниз, влево и вправо. Таким образом, нарисованная линия неизбежно получится неровной, и единственный способ сделать ее красивой и гладкой — использовать экран высокого разрешения как можно с более мелкими пикселями. Заметим, однако, что человеческий глаз, будучи по сути еще одной «машиной», также имеет ограниченную способность к восприятию изображения, зависящую от количества и типа рецепторных клеток в сетчатке. Таким образом, экраны с высоким разрешением могут обманывать человеческий мозг, заставляя его поверить, что линии, состоящие из пикселей, визуальны ровные. На самом деле они всегда неровные.

Процедура построения линии от $(x1, y1)$ до $(x2, y2)$ начинается с отрисовки пикселя $(x1, y1)$, после чего происходит зигзагообразное перемещение в направлении $(x2, y2)$. Псевдокод этого алгоритма показан на иллюстрации 12.7.

Поскольку в данном алгоритме на каждую итерацию цикла приходится две операции деления, то можно сделать вывод, что этот алгоритм неэффективен и неточен. Первое очевидное улучшение заключается в замене условия $b / a > dy / dx$ на эквивалентное $a \cdot dy < b \cdot dx$, требующее только целочисленного умножения. Анализ последнего условия показывает, что его можно проверить и без умножения. Как показано в улучшенном алгоритме на иллюстрации 12.7, это можно сделать эффективно при помощи переменной, обновляющей значение $(a \cdot dy - b \cdot dx)$ при каждом увеличении a или b .

Время работы такого алгоритма отрисовки линии составляет $O(n)$, где n — количество пикселей вдоль рисуемой линии. Алгоритм

использует только операции сложения и вычитания, и его можно эффективно реализовать как программно, так и аппаратно.

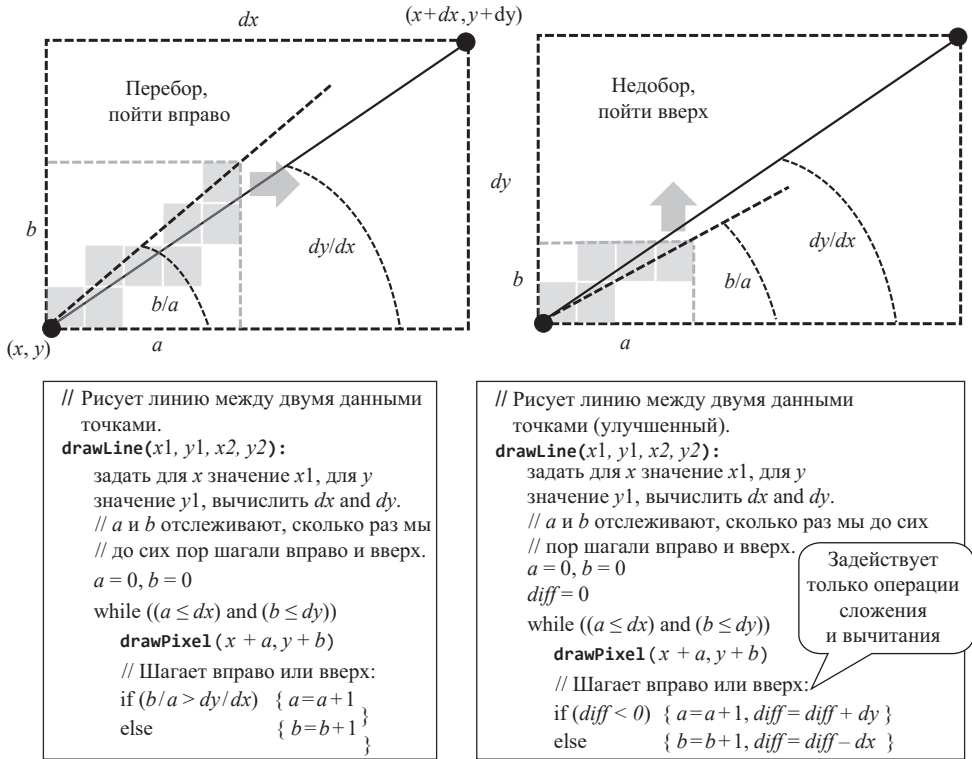
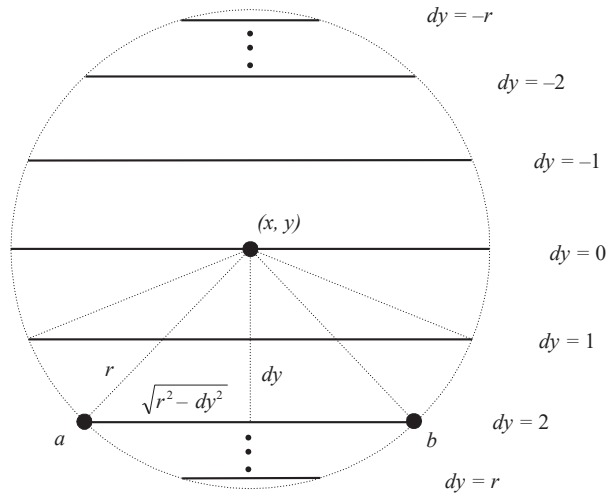


Иллюстрация 12.7. Алгоритм рисования линии: базовая версия (внизу слева) и улучшенная версия (внизу справа).

Рисование круга (drawCircle). На иллюстрации 12.8 представлен алгоритм, использующий три уже реализованные нами процедуры: умножение, вычисление квадратного корня и рисование линий.

Алгоритм основан на отрисовке последовательности горизонтальных линий (подобных типичной линии ab на иллюстрации), по одной на каждую строку в диапазоне от $y - r$ до $y + r$. Поскольку r задается в пикселях, алгоритм в итоге рисует линию в каждой строке вдоль диаметра окружности сверху вниз, в результате чего получается полностью заполненный круг. Нарисовать одну лишь окружность (контур) при необходимости можно посредством простой настройки.



Точка $a = (x - \sqrt{r^2 - dy^2}, y + dy)$

Точка $b = (x + \sqrt{r^2 - dy^2}, y + dy)$

```
// Рисует заполненный круг радиусом r с центром в точке (x, y).
```

```
drawCircle(x, y, r):
```

```
  for each  $dy = -r$  to  $r$  do:
```

```
    drawLine(( $x - \sqrt{r^2 - dy^2}$ ,  $y + dy$ ), ( $x + \sqrt{r^2 - dy^2}$ ,  $y + dy$ ))
```

Иллюстрация 12.8. Алгоритм рисования круга.

12.1.5. Вывод символов

Для разработки возможности вывода символа, превратим сначала наш физический, ориентированный на пиксели экран в логический, ориентированный на символы, подходящий для вывода фиксированных растровых изображений отдельных установленных символов. Рассмотрим для примера физический экран, имеющий 256 строк по 512 пикселей в каждой. Если для отрисовки одного символа выделить область размером в 11 строк на 8 столбцов, то на этом экране можно будет отобразить 23 строки по 64 символа в каждой, при этом три дополнительных строки пикселей останутся неиспользованными.

Шрифты: наборы символов, которые используют компьютеры, делятся на *печатаемые* и *непечатаемые* подмножества. Для каждого печатаемого символа в наборе символов Nask (см. приложение 5) мы в меру своих ограниченных способностей постарались составить растровое изображение размером в 11 строк на 8 столбцов. Все вместе эти изображения называются *шрифтом* (*font*). На иллюстрации 12.9 показано, как в нашем шрифте отображается заглавная буква N. Для обработки интервалов между символами каждое изображение символа включает пространство шириной по крайней мере в один пиксель перед следующим символом в строке и пространство шириной по крайней мере в один пиксель между соседними строками (точный интервал зависит от размера и характера линий отдельных символов). Шрифт Nask состоит из девяноста пяти таких растровых изображений, по одному для каждого печатаемого символа в наборе символов Nask.

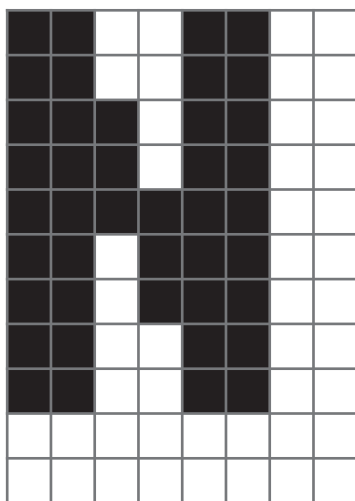


Иллюстрация 12.9. Пример растрового изображения (*bitmap*) символа.

Дизайн шрифтов — древнее, но до сих пор активно развивающееся искусство. Некоторые старые шрифты — ровесники самого искусства письма, но дизайнеры, желающие заявить о себе или решить какую-либо техническую или функциональную задачу, постоянно разрабатывают новые. В нашем случае ограничения небольшого физического

экрана, с одной стороны, и желание отобразить разумное количество символов в каждой строке, с другой, привели к прагматическому выбору экономной площади изображения в 11 на 8 пикселей. Соображения экономии заставили нас разработать грубый, но тем не менее неплохо выполняющий свои задачи шрифт.

Курсор: символы обычно отображаются один за другим слева направо до конца строки. Рассмотрим для примера программу, в которой за командой `print("a")` следует (возможно, не сразу) команда `print("b")`. Это означает, что программа хочет вывести на экран `ab`. Для поддержки непрерывной печати пакет отображения символов поддерживает глобальный *курсор*, отслеживающий место на экране, где должен быть нарисован следующий символ. Информация о курсоре состоит из указаний на номера столбца и строки, например, `cursor.col` и `cursor.row`. После отображения очередного символа осуществляется операция `cursor.col++`. В конце строки выполняются операции `cursor.row++` и `cursor.col = 0`. При достижении нижней границы экрана возникает вопрос, что же делать дальше. Возможны два варианта действий: выполнить прокрутку или очистить экран и начать отображать символы сначала, установив курсор в положение `(0, 0)`.

Напомним, что мы описали схему отображения на экране отдельных символов. Из этой базовой возможности естественным образом вытекает возможность записи других типов данных: строки записываются символ за символом, а числа сначала преобразуются в строки, а затем записываются как строки.

12.1.6. Ввод с клавиатуры

Ввод поступающих с клавиатуры данных более сложен, чем кажется на первый взгляд. Рассмотрим для примера команду `let name = Keyboard.readLine("введите ваше имя: ")`. Выполнение функции `readLine` по определению зависит от действий довольно непредсказуемого объекта — человека. Функция не завершится, пока пользователь не нажмет несколько клавиш на клавиатуре с клавишей

ENTER в конце. Проблема тут в том, что время, в течение которого человек нажимает на отдельную клавишу, прежде чем отпустить ее, непредсказуемо, как и непредсказуемо время набора всего текста, в течение которого человек может сделать перерыв и пойти пить кофе. Кроме того, люди любят удалять и перепечатывать символы. Реализация функции `readLine` должна учитывать все эти детали.

В этом разделе управление вводом с клавиатуры описывается на трех уровнях абстракции: 1) определение того, какая клавиша в данный момент нажата на клавиатуре, 2) захват одного вводимого символа и 3) захват нескольких введенных символов.

Прослушивание клавиатуры

```
// Возвращает код символа
// нажатой в настоящий момент клавиши
// или 0, если клавиша не нажата.
```

keyPressed():

```
// использует Memory.peek
```

Получение символа

```
//Дождется, пока нажатая клавиша не будет отпущена;
// отображает символ на экране, перемещает
// курсор и возвращает код символа.
```

readChar():

```
отобразить курсор
```

```
// ждет нажатия клавиши:
```

```
while (keyPressed() == 0)
```

```
do nothing (ничего)
```

```
c = код нажимаемой в данный момент клав
```

```
// ждет, пока клавиша не будет отпущена:
```

```
while (keyPressed() != 0)
```

```
do nothing (ничего)
```

```
отобразить c в текущей локации курсора
```

```
переместить курсор
```

```
return c
```

Получение строки

```
// Отображает сообщение message на экране,
// получает следующую строку (до ввода символа
// новой строки newLine) с клавиатуры
// и возвращает ее значение в виде строки.
```

readLine(message):

```
отобразить message
```

```
str = пустая строка
```

```
repeat
```

```
c = readChar()
```

```
if (c == newLine):
```

```
display newLine
```

```
return str
```

```
else if (c == backSpace): (символ возврата)
```

```
удалить последний символ из str
```

```
передвинуть курсор назад
```

```
else
```

```
str.appendChar(c)
```

```
return str
```

Иллюстрация 12.10. Обработка ввода с клавиатуры.

Обнаружение ввода с клавиатуры (`keyPressed`): определение нажимаемой в данный момент клавиши — специфическая операция, зависящая от интерфейса клавиатуры. В компьютере Nask клавиатура постоянно обновляет 16-битный регистр памяти, адрес которого хранится в указателе с именем KBD. Правило взаимодействия выглядит следующим образом: если на клавиатуре в данный момент нажата

некая клавиша, то этот регистр содержит код символа клавиши (набор символов `Mask` приведен в приложении 5); в противном случае он содержит 0. Это правило используется для реализации функции `keyPressed`, показанной на иллюстрации 12.10.

Чтение одного символа (`readChar`): длительность промежутка времени между нажатием клавиши и последующим ее отпусканием непредсказуема. Следовательно, мы должны написать код, который устранил эту неопределенность. Кроме того, мы хотим предусмотреть обратную связь, то есть передавать информацию о том, когда какие клавиши были нажаты (то, что вы, вероятно, привыкли воспринимать как должное). Кроме того, хочется отобразить графический курсор в том месте экрана, где будет происходить ввод следующего символа, и после нажатия какой-либо клавиши показывать введенный символ в виде его растрового изображения на экране в месте расположения курсора. Все эти действия реализуются функцией `readChar`.

Чтение строки (`readLine`): строка из нескольких введенных пользователем символов считается окончательной после нажатия клавиши `ENTER`, передающей символ новой строки `newLine`. До нажатия клавиши `ENTER` у пользователя должна быть возможность удалять и повторно набирать символы. Все эти действия выполняет функция `readLine`. Как обычно, наши решения по обработке ввода основаны на каскадной серии абстракций: программа высокого уровня полагается на абстракцию `readLine`, которая полагается на абстракцию `readChar`, которая полагается на абстракцию `keyPressed`, которая полагается на абстракцию `Memory.peek`, которая полагается на аппаратное обеспечение.

12.2. Спецификация ОС Jack

В предыдущем разделе были представлены алгоритмы, решающие различные классические задачи операционной системы. В этом разделе мы переходим к формальному описанию одной конкретной

операционной системы — ОС Jack. Операционная система Jack состоит из восьми классов.

- `Math`: обеспечивает математические операции.
- `String`: реализует тип `String`.
- `Array`: реализует тип `Array`.
- `Memory`: осуществляет операции с памятью.
- `Screen`: осуществляет вывод графики на экран.
- `Output`: осуществляет вывод символов на экран.
- `Keyboard`: обрабатывает ввод с клавиатуры.
- `Sys`: обеспечивает сервисы, связанные с выполнением программ.

Полный API ОС приведен в приложении 6. Этот API можно рассматривать как спецификацию ОС. В следующем разделе описывается, как его можно реализовать с помощью алгоритмов, представленных в предыдущем разделе.

12.3. Реализация

Каждый класс ОС представляет собой набор процедур (конструкторов, функций и методов). Большинство процедур ОС просты в реализации и здесь не рассматриваются. Остальные процедуры ОС основаны на алгоритмах, представленных в разделе 12.2. При реализации данных процедур можно воспользоваться следующими советами и рекомендациями.

Функции `init`: некоторые классы ОС для поддержки реализации своих процедур используют определенные структуры данных. Для каждого такого класса `OSClass` эти структуры данных можно объявлять статически на уровне класса и инициализировать функцией, которую мы условно называем `OSClass.init`. Функции `init` предназначены для внутренних целей и не документированы в API ОС.

Math

multiply (умножение): на каждой итерации i алгоритма *умножения* (см. иллюстрацию 12.1) извлекается i -й бит второго множителя. Мы предлагаем оформить эту операцию в виде вспомогательной функции `bit(x, i)`, которая возвращает `true`, если i -й бит целого числа x равен 1, и `false` в противном случае. Функцию `bit(x, i)` можно легко реализовать с помощью операций сдвига. К сожалению, Jack не поддерживает сдвиг. Вместо этого можно определить фиксированный статический массив длины 16, допустим, `twoToThe`, и присвоить каждому элементу i значение 2 в степени i . Затем данный массив можно использовать для поддержки реализации `bit(x, i)`. Массив `twoToThe` создается функцией `Math.init`.

divide (деление): алгоритмы *умножения* и *деления*, представленные на иллюстрациях 12.1 и 12.2, предназначены для работы с неотрицательными целыми числами. Знаковые числа можно обрабатывать, применяя алгоритмы к абсолютным значениям и соответствующим образом устанавливая знак возвращаемых значений. Для алгоритма *умножения* это не требуется: поскольку множители представлены в виде дополнения до двух (с дополнительным кодом), их произведение будет правильным без дополнительных действий.

В алгоритме *деления* y умножается на коэффициент 2 до тех пор, пока не станет верным условие $y > x$. Но при этом y может переполниться. Переполнение можно обнаружить, проверив, когда y станет отрицательным.

sqrt (квадратный корень): в алгоритме вычисления *квадратного корня* (иллюстрация 12.3) значение $(y + 2^i)^2$ может переполниться, что приведет к неверному отрицательному результату. Эту проблему можно решить, эффективно изменив логику алгоритма `if` следующим образом: если $(y + 2^i)^2 \leq x$ и $(y + 2^i)^2 > 0$, то $y = y + 2^i$.

String

Все строковые константы, которые появляются в программе Jack, реализуются как объекты класса `String`, API которого задокументирован в приложении 6. В частности, каждая строка реализуется как объект, состоящий из массива значений `char`, со свойством `maxLength`, которое содержит максимальную длину строки, и свойством `length`, содержащим фактическую длину строки.

Рассмотрим для примера команду `let str="scooby"`. Обработывая это высказывание, компилятор вызывает конструктор `String`, создающий массив `char` с `maxLength=6` и `length=6`. Если позже мы вызовем метод `String str.eraseLastChar()`, длина массива станет равной 5, а строка превратится в `"scoob"`. В общем случае элементы массива, превышающие длину, не считаются частью строки.

Что должно происходить при попытке добавить символ в строку, длина которой равна `maxLength`? Спецификация ОС не определяет этот вопрос: класс `String` может изменять размер массива, а может оставлять его прежним — это остается на усмотрение отдельных реализаций ОС.

intValue, setInt: эти подпрограммы можно реализовать с помощью алгоритмов, показанных на иллюстрации 12.4. Обратите внимание, что ни один из алгоритмов не обрабатывает отрицательные числа — эту деталь предлагается реализовать самостоятельно.

newLine, backSpace, doubleQuote: как видно из приложения 5, коды этих символов равны 128, 129 и 34.

Остальные методы `String` можно реализовать, непосредственно оперируя массивом `char` и полем `length`, которое характеризует каждый объект `String`.

Array

new: несмотря на свое название, эта процедура не конструктор, а скорее функция. Поэтому ее реализация должна выделять место в памяти

для нового массива посредством явного вызова функции `OSMemory.alloc`.

dispose: этот `void`-метод вызывается высказываниями типа `do arr.dispose()`. Реализация `dispose` освобождает память массива, вызывая функцию `OSMemory.deAlloc`.

Memory

peek, poke: эти функции обеспечивают прямой доступ к памяти хоста. Как же осуществить такой низкоуровневый доступ с помощью языка высокого уровня Jack? Оказывается, в языке Jack есть некая «лазейка», позволяющая получить полный контроль над памятью хост-компьютера. Эту лазейку можно использовать для реализации `Memory.peek` и `Memory.poke` обычными программными средствами языка Jack.

Лазейка основана на аномальном использовании ссылочной переменной (указателя). Jack — слабо типизированный язык; среди прочих особенностей он не запрещает программисту присваивать константу ссылочной переменной. Эту константу можно рассматривать как абсолютный адрес памяти. Когда ссылочная переменная оказывается массивом, эта схема обеспечивает индексированный доступ к каждому слову в оперативной памяти хоста, как показано на иллюстрации 12.11.

```
// Создает "прокси" RAM:  
var Array memory;  
let memory \ 0; // Без проблем...  
...  
// Получает значение RAM по адресу i:  
let x \ memory[i];  
...  
// Устанавливает значение RAM по адресу i:  
let memory[i] \ 17;  
...
```

Иллюстрация 12.11. Лазейка, позволяющая получить полный контроль над ОЗУ (RAM) хоста из Jack.

После первых двух строк кода основание (база) массива `memory` указывает на первый адрес в оперативной памяти компьютера (RAM адрес 0). Чтобы получить или установить значение ячейки оперативной памяти, физический адрес которой равен i , достаточно манипулировать элементом массива `memory[i]`. Это заставит компилятор манипулировать ячейкой оперативной памяти, адрес которой равен $0 + i$, что нам и нужно.

Массивам `Jack` выделяется место в куче не во время компиляции, а во время выполнения, если вызывается функция `new` массива. Обратите внимание, что если бы процедура `new` была конструктором, а не функцией, то компилятор и ОС выделили бы новый массив по какому-то непонятному адресу ОЗУ, недоступному для нашего контроля. Как и многие классические «хаки», этот трюк работает потому, что мы используем переменную массива, не инициализируя ее надлежащим образом, как это обычно делается при использовании массивов.

Массив `memory` можно объявлять на уровне класса и инициализировать функцией `Memory.init`. После выполнения этого трюка реализация `Memory.peek` и `Memory.poke` станет тривиальным делом.

`alloc`, `deAlloc`: эти функции можно реализовать одним из алгоритмов, показанных на иллюстрациях 12.5а и 12.5б. Для реализации `Memory.deAlloc` можно использовать либо метод «лучшего соответствия», либо метод «первого подходящего».

Согласно стандартному отображению ВМ на платформе `Hack` (см. раздел 7.4.1), стек должен отображаться на адреса оперативной памяти с 256 по 2047. Таким образом, куча может начинаться с адреса 2048.

Для реализации связанного списка `freeList` класс `Memory` может объявлять и поддерживать статическую переменную `freeList`, как показано на иллюстрации 12.12. Хотя `freeList` инициализируется значением `heapBase` (2048), возможно, что после нескольких операций `alloc` и `deAlloc` `freeList` станет каким-то другим адресом в памяти, как показано на иллюстрации.

Для повышения эффективности рекомендуется написать код `Jack`, управляющий связанным списком `freeList` непосредственно

в оперативной памяти, как показано на иллюстрации 12.12. Связанный список можно инициализировать функцией `Memory.init`.

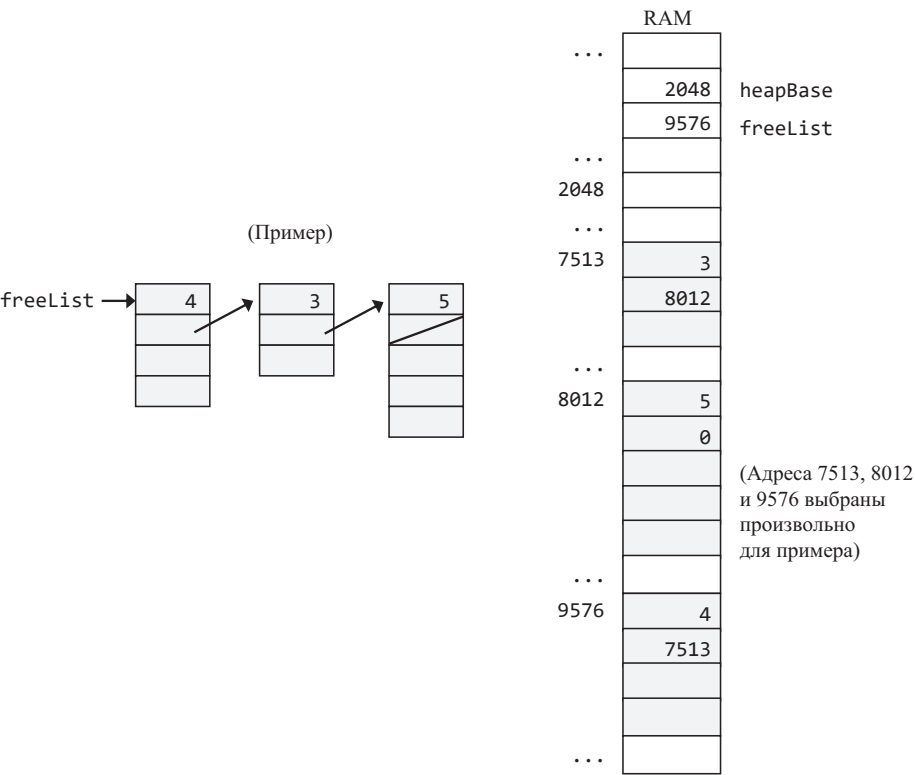


Иллюстрация 12.12. Логическое представление (слева) и физическая реализация (справа) связанного списка, поддерживающего динамическое распределение памяти.

Screen

Класс `Screen` поддерживает *текущий цвет*, который используется всеми функциями рисования класса. Текущий цвет можно представить в виде статической булевой переменной.

drawPixel: рисование пикселя на экране можно выполнить с помощью `Memory.peek` и `Memory.poke`. Согласно схеме карты памяти

экрана платформы Hack, пиксель в столбце col и строке row ($0 \leq col \leq 511$, $0 \leq row \leq 255$) отображается на бит $col \% 16$ ячейки памяти $16384 + row \cdot 32 + col / 16$. Перерисовка одного пикселя требует изменения одного (и только одного этого) бита в слове.

drawLine: базовый алгоритм, показанный на иллюстрации 12.7, потенциально может привести к переполнению. Однако эту проблему устраняет улучшенная версия алгоритма.

Для рисования линий, простирающихся в четырех возможных направлениях, придется обобщить некоторые аспекты алгоритма. Следует помнить о том, что начало координат экрана $(0, 0)$ находится в левом верхнем углу. Поэтому в вашей реализации процедуры `drawLine`, возможно, будет нужно поменять местами некоторые операции сложения и вычитания.

Особые, но часто встречающиеся случаи рисования вертикальных и горизонтальных линий, когда $dx = 0$ или $dy = 0$, лучше рассматривать отдельно и использовать для них отдельную оптимизированную реализацию.

drawCircle: алгоритм, показанный на иллюстрации 12.8, потенциально может привести к переполнению. Разумным решением будет ограничить радиус окружности максимальным значением 181.

Output

Класс `Output` — это библиотека функций для отображения символов. Класс предполагает наличие символьно-ориентированного экрана, состоящего из 23 строк (с индексами $0 \dots 22$, сверху вниз) по 64 символа в каждой (с индексами $0 \dots 63$, слева направо). Самый верхний левый участок имеет индекс $(0, 0)$. Видимый курсор, реализованный в виде небольшого заполненного квадрата, указывает, где будет отображаться следующий символ. Каждый символ отображается на экране в виде прямоугольного изображения высотой 11 пикселей и шириной 8 пикселей (включая поля для интервалов между символами и между строками). Совокупность всех изображений символов называется *шрифтом*.

Реализация шрифта: разработка и реализация шрифта для набора символов Nask (приложение 5) — это кропотливая работа, требующая как некоторых художественных умений, так и усердия в рутинной работе по реализации. Полученный шрифт являет собой набор из девяноста пяти прямоугольных растровых изображений, каждое из которых представляет собой печатаемый символ.

Шрифты обычно хранятся во внешних файлах, загружаемых и используемых пакетом отображения символов по мере необходимости. В нашем практическом курсе «От Nand до “Тетриса”» шрифт встроен в класс `OS Output`. Для каждого печатаемого символа мы определяем массив, в котором хранится его растровая (битовая) карта. Массив состоит из 11 элементов, каждый из которых соответствует строке из 8 пикселей. Точнее, каждому элементу массива j присваивается целочисленное значение, двоичное представление которого кодирует 8 пикселей, появляющихся в j -й строке битовой карты символа. Вместе с этим мы определяем статический массив размером 127, индексные значения которого от 32 до 126 соответствуют кодам печатаемых символов в наборе символов Nask (элементы 0... 31 не используются). Затем каждому i -му элементу этого массива присваивается массив из 11 элементов, представляющий собой растровое изображение символа, код которого равен i (надеюсь, что вы уже убедились в трудоемкости этой задачи).

Материалы проекта 12 включают каркасный класс `Output`, содержащий код `Jack`, который выполняет всю описанную выше работу по реализации. В целом он реализует шрифт из девяноста пяти символов, кроме одного, разработать и реализовать который мы предлагаем вам в качестве упражнения. Этот код можно активировать функцией `Output.init`, которая также может инициализировать курсор.

printChar: отображает символ в месте расположения курсора и перемещает курсор на один столбец вперед. Чтобы отобразить символ в месте (row, col) , где $0 \leq row \leq 22$ и $0 \leq col \leq 63$, мы записываем битовую (растровую) карту символа на область пикселей в диапазоне от $11 \cdot row$ до $11 \cdot row + 10$ и от $8 \cdot col$ до $8 \cdot col + 7$.

printString: можно реализовать с помощью последовательности вызовов `printChar`.

printInt: можно реализовать посредством преобразования целого числа в строку и последующей печати строки.

Keyboard

В соответствии с организацией памяти компьютера Hack (см. раздел 5.2.6) *карта памяти* клавиатуры представляет собой один 16-битный регистр памяти, расположенный по адресу 24576.

keyPressed: можно легко реализовать с помощью `Memory.peek()`.

readChar, readString: можно реализовать, следуя алгоритмам на иллюстрации 12.10.

readInt: можно реализовать посредством чтения строки и преобразования ее в значение `int` с помощью метода `String`.

Sys

wait: эта функция должна подождать заданное количество миллисекунд и осуществить возврат. Ее можно реализовать, написав цикл, который выполняется примерно `duration` миллисекунд, но для точного вычисления длительности придется засечь время на конкретном компьютере, потому что эта константа варьируется от одного процессора к другому. Получившуюся в результате функцию `Sys.wait()` нельзя будет перенести на другие компьютеры. Ее можно сделать переносимой, запустив еще одну функцию конфигурации, которая будет устанавливать различные константы, отражающие аппаратные характеристики хост-платформы, но для проекта «От Nand до “Тетриса”» это не нужно.

halt: можно реализовать посредством входа в бесконечный цикл.

init: согласно спецификации языка Jack (см. раздел 9.2.2), программа на языке Jack представляет собой набор из одного или нескольких классов. Один класс должен называться `Main` и включать функцию `main`. Чтобы начать выполнение программы, необходимо вызвать функцию `Main.main`.

Операционная система также представляет собой набор скомпилированных классов Jack. При загрузке компьютера запускается операционная система, которая, в свою очередь, запускает главную программу. Эта цепочка команд реализуется следующим образом. В соответствии со стандартным отображением ВМ на платформе Hack (раздел 8.5.2) транслятор ВМ пишет загрузочный код (на машинном языке), который вызывает функцию `OS.Sys.init`. Данный загрузочный код хранится в ПЗУ (ROM), начиная с адреса 0. При перезагрузке компьютера программный счетчик обнуляется, начинает выполняться загрузочный код и вызывается функция `Sys.init`.

Исходя из вышесказанного, `Sys.init` должна выполнять две задачи: вызывать все функции `init` других классов ОС, а затем вызывать `Main.main`.

С этого момента пользователь переходит во власть прикладной программы, а наше путешествие «От Nand до “Тетриса”» подходит к концу. Надеемся, оно вам понравилось!

12.4. Проект

Задача: реализовать операционную систему, описанную в этой главе.

Контракт: реализуйте операционную систему в Jack и протестируйте ее с помощью программ и сценариев тестирования, описанных ниже. Каждая тестовая программа использует подмножество сервисов ОС. Каждый из классов ОС можно реализовывать и тестировать изолированно в любом порядке.

Ресурсы: основной необходимый инструмент — это язык Jack, на котором вы будете разрабатывать ОС. Еще вам понадобятся прилагаемый

компилятор Jack для компиляции вашей реализации ОС, а также прилагаемые тестовые программы, тоже написанные на Jack. Наконец, вам понадобится прилагаемый эмулятор VM — платформа, на которой будут выполняться тесты. В папке `projects/12` на вашем компьютере содержатся восемь файлов классов каркасной ОС с именами `Math.jack`, `String.jack`, `Array.jack`, `Memory.jack`, `Screen.jack`, `Output.jack`, `Keyboard.jack` и `Sys.jack`. Каждый файл содержит сигнатуры всех подпрограмм класса. Ваша задача — дописать недостающие реализации.

Эмулятор VM: разработчики операционных систем часто сталкиваются с проблемой «курицы и яйца»: как протестировать класс ОС в изоляции, если этот класс использует сервисы других, еще не разработанных классов ОС? Как выяснилось, для поддержки модульного тестирования ОС, по одному классу за раз, идеально подходит эмулятор виртуальной машины.

В частности, эмулятор виртуальной машины имеет исполняемую версию ОС, написанную на Java. Когда в загруженном коде виртуальной машины обнаруживается вызов команды `f00`, эмулятор действует следующим образом. Если функция VM с именем `f00` существует в загруженной кодовой базе, эмулятор выполняет ее код VM. В противном случае эмулятор проверяет, является ли `f00` одной из встроенных функций ОС. Если да, то выполняется встроенная реализация `f00`. Такое соглашение идеально подходит для описанной далее стратегии тестирования.

План тестирования

Папка `projects/12` на вашем компьютере содержит восемь тестовых папок, названных `MathTest`, `MemoryTest` и т. д. для тестирования каждого из восьми классов ОС `Math`, `Memory` и т. д. Каждая папка содержит программу Jack, предназначенную для тестирования сервисов соответствующего класса ОС. Некоторые папки содержат

тестовые скрипты и файлы сравнения, а некоторые — только файл или файлы `.jack`. Для тестирования своей реализации класса ОС `Xxx.jack` вы можете поступить следующим образом.

- Изучите прилагаемый код тестовой программы `XxxTest/*.jack`.
- Поместите разработанный вами класс ОС `Xxx.jack` в папку `XxxTest`.
- Скомпилируйте папку с помощью прилагаемого компилятора `Jack`. В результате будет транслирован файл класса ОС и файл или файлы `.jack` тестовой программы в соответствующие файлы `.vm`, хранящиеся в той же папке.
- Если папка содержит тестовый сценарий `.tst`, загрузите сценарий в эмулятор виртуальной машины; в противном случае загрузите в эмулятор виртуальной машины папку.
- Следуйте конкретным рекомендациям по тестированию, приведенным ниже для каждого класса ОС.

Memory, Array, Math: три папки, в которых тестируются эти классы, включают тестовые сценарии и файлы сравнения. Каждый тестовый сценарий начинается с команды `load`. Эта команда загружает все файлы `.vm` в текущей папке в эмулятор виртуальной машины. Следующие две команды в каждом тестовом сценарии создают выходной файл и загружают прилагаемый файл сравнения. Далее сценарий тестирования выполняет несколько тестов, сравнивая результаты тестов с результатами, указанными в файле сравнения. Ваша задача — убедиться, что эти сравнения завершились успешно.

Обратите внимание, что прилагаемые тестовые программы не тестируют полностью `Memory.alloc` и `Memory.deAlloc`. Полное тестирование этих функций управления памятью требует проверки внутренних деталей реализации, не видимых при тестировании на уровне пользователя. При желании можно протестировать данные функции с помощью пошаговой отладки и осмотра состояния оперативной памяти хоста.

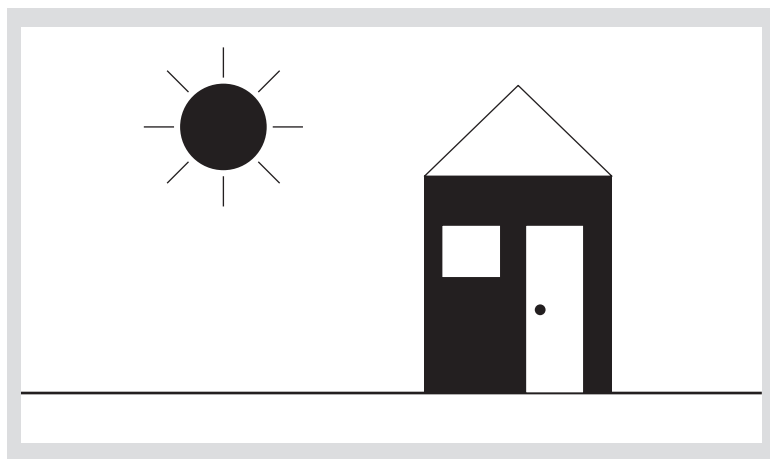
String: выполнение прилагаемой тестовой программы должно дать следующий результат.

```
new,appendChar: abcde
setInt: 12345
setInt: -32767
length: 5
charAt[2]: 99
setCharAt(2,'-'): ab-de
eraseLastChar: ab-d
intValue: 456
intValue: -32123
backSpace: 129
doubleQuote: 34
newLine: 128
```

Output: выполнение прилагаемой тестовой программы должно дать следующий результат.

```
A
0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
!#$%&'()*+,-./:;<=>?@[^_`{|}~"
-12346789
C
```

Screen: выполнение прилагаемой тестовой программы должно дать следующий результат.



Keyboard: этот класс ОС тестируется тестовой программой, позволяющей пользователю взаимодействовать с ней. Для каждой функции класса `Keyboard` (`keyPressed`, `readChar`, `readLine`, `readInt`) программа предлагает пользователю нажать некоторые клавиши. Если функция ОС реализована правильно и запрошенные клавиши нажаты, программа выводит `ok` и переходит к тестированию следующей функции ОС. В противном случае программа повторяет запрос. Если все запросы завершаются успешно, программа печатает `Test ended successfully`. В этот момент на экране должно появиться следующее сообщение.

```
keyPressed test:
Please press the 'Page Down' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3': 3
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter: JACK
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: -32123
ok
Test completed successfully
```

Sys

Прилагаемый файл `.jack` тестирует функцию `Sys.wait`. Программа просит пользователя нажать клавишу (любую) и ждет две секунды, используя вызов `Sys.wait`. Затем она печатает сообщение на экране. Убедитесь, что время, которое проходит между нажатием клавиши и появлением на экране сообщения, составляет около двух секунд.

Функция `Sys.init` в явном виде не тестируется, но вспомните, что она выполняет все необходимые инициализации ОС, а затем вызывает функцию `Main.main` каждой тестовой программы. Поэтому можно предположить, что, если `Sys.init` реализована некорректно, ничего правильно запуститься не будет.

Полный тест

После успешного тестирования каждого класса ОС по отдельности протестируйте всю реализацию ОС с помощью представленной ранее игры «Pong». Исходный код Pong доступен в папке `projects/11/Pong`. Поместите восемь файлов ОС `.jack` в папку Pong и скомпилируйте папку с помощью прилагаемого компилятора Jack. Затем

загрузите папку Pong в эмулятор ВМ, запустите игру и убедитесь, что она работает так, как ожидается.

12.5. Перспектива

В этой главе было описано подмножество основных сервисов (служб), которые можно найти в большинстве операционных систем: управление памятью, управление устройствами ввода-вывода, не реализованные в аппаратуре математические операции и реализация абстрактных типов данных, таких как абстракция строк. Мы решили назвать эту стандартную библиотеку программного обеспечения *операционной системой*, чтобы отразить две ее основные функции: инкапсулирование особенностей аппаратного обеспечения и некоторых не реализованных на ранних стадиях процедур в прозрачные программные сервисы и предоставление компиляторам и прикладным программам возможности использовать эти сервисы через понятные интерфейсы. Однако тому, что мы называем ОС, еще очень далеко до настоящих промышленных операционных систем.

Прежде всего в нашей ОС отсутствуют некоторые из основных сервисов, наиболее тесно связанных с операционными системами. Например, наша ОС не поддерживает ни многопоточность, ни многопроцессорность, в то время как ядра большинства операционных систем занимаются именно этим. Наша ОС не поддерживает устройства хранения данных, в то время как именно абстракция файловой системы поддерживает основные хранилища данных, с которыми работают операционные системы. В нашей ОС нет ни интерфейса командной строки (как в оболочке Unix), ни графического интерфейса, состоящего из окон и меню. Но ведь именно интерфейс операционной системы ожидают пользователи увидеть в первую очередь, и именно с ним они взаимодействуют. В нашей ОС отсутствуют множество других сервисов, обычно встречающихся в операционных системах, таких как сервисы безопасности, обмена данными и др.

Еще одно заметное отличие заключается в довольно либеральном подходе к вызову операций ОС. Некоторые операции ОС, например

реек и роке, дают программисту полный доступ к ресурсам компьютера. Очевидно, что непреднамеренное или злонамеренное использование таких функций может привести к хаосу. Поэтому многие службы ОС считаются привилегированными и доступ к ним требует более сложного механизма безопасности, чем простой вызов функции. В отличие от этого, в платформе Hack нет разницы между кодом ОС и кодом пользователя, а службы операционной системы работают в том же пользовательском режиме, что и прикладные программы.

С точки зрения эффективности алгоритмы, которые мы представили для умножения и деления, были стандартными. Эти алгоритмы или их варианты обычно реализуются аппаратно, а не программно. Их время работы составляет $O(n)$ операций сложения. Поскольку сложение двух n -битных чисел требует $O(n)$ битовых операций (вентилей в аппаратуре), в итоге эти алгоритмы требуют $O(n^2)$ битовых операций. Существуют алгоритмы умножения и деления, время работы которых асимптотически значительно быстрее, чем $O(n^2)$, и для большого числа битов эти алгоритмы более эффективны. Подобным образом для ускорения графики оптимизированные версии представленных нами геометрических операций, таких как рисование линий и окружностей, часто реализуются в специальном аппаратном обеспечении.

Как это верно в отношении любой аппаратной и программной системы, разрабатываемой в рамках практического курса «От Nand до “Тетриса”», наша цель состояла не в том, чтобы предоставить полное решение, удовлетворяющее всем желаниям и потребностям. Мы скорее стремились создать рабочую реализацию, опираясь на твердое понимание основ системы, и предлагать способы ее дальнейшего расширения. Некоторые из этих дополнительных проектов расширения упомянуты в следующей, последней главе книги.

13. Веселье продолжается

Мы не оставим поиск, мы придем в конце в то место, из которого ушли, но место не узнаем.

— Т. С. Элиот (1888–1965)

Поздравляем! Мы завершили построение полной вычислительной системы, начав с самых основ. Надеемся, вам понравилось это захватывающее путешествие. Но позвольте авторам этой книги поделиться с вами одним секретом: на наш взгляд, еще большее удовольствие мы получили от написания книги, ведь нам самим пришлось проектировать всю эту вычислительную систему, а проектирование бывает самой «веселой» частью любого проекта. Мы не сомневаемся в том, что некоторые из вас, самые отважные и любознательные, тоже захотят принять участие в проектировании. Возможно, вам захотелось усовершенствовать архитектуру или уже появились какие-то идеи по добавлению новых функций, а может быть, вы подумали о том, чтобы разработать более широкую систему. А потом, возможно, вам захочется самим взять в руки штурвал и решать, куда двигаться дальше.

Улучшить, оптимизировать или расширить можно почти каждый аспект системы Jack/Hack. Например, переписывая части своих реализаций ассемблера, компилятора и ОС, вы сможете изменить и дополнить язык ассемблера, язык Jack и операционную систему. Другие изменения, скорее всего, потребуют модификации поставляемых программных инструментов. Например, если вы измените спецификацию аппаратного обеспечения или спецификацию виртуальной

машины, то вам, скорее всего, придется модифицировать соответствующие эмуляторы. Или если хотите добавить в компьютер Nask больше устройств ввода или вывода, вам, вероятно, придется смоделировать их, написав новые встроенные микросхемы.

Чтобы обеспечить полную гибкость модификации и расширений, мы сделали исходный код всех поставляемых инструментов общедоступным. Весь наш код на сто процентов написан в Java, за исключением некоторых пакетных файлов, используемых для запуска программы на некоторых платформах. Программное обеспечение и документация к нему доступны по адресу: www.nand2tetris.org. Вы вправе изменять и расширять все наши инструменты так, как сочтете нужным для реализации своих идей, а затем по желанию можете делиться ими с другими пользователями. Мы надеемся, что наш код написан и задокументирован достаточно хорошо, чтобы работа над модификациями приносила вам удовольствие. В частности, отметим, что прилагаемый симулятор аппаратуры имеет простой и хорошо задокументированный интерфейс для добавления новых встроенных микросхем. Его интерфейс можно использовать для расширения моделируемой аппаратной платформы, например, для добавления запоминающих устройств или устройств связи.

Конечно, всех ваших идей мы предвидеть не сумеем, но можем кратко перечислить некоторые из тех, о которых задумывались сами.

Аппаратные реализации

Описанные в книге аппаратные модули были реализованы либо на языке HDL, либо в виде прилагаемых исполняемых программных модулей. Однако в какой-то момент HDL-проекты обычно переносятся в физическую форму и реализуются «в кремнии», становясь настоящими компьютерами. Разве не было бы здорово заставить Nask или Jack работать на настоящей аппаратной платформе, состоящей из атомов, а не битов?

При этом возможны разные подходы. С одной стороны, можно попытаться реализовать платформу Nask на ППВМ (программируемой пользователем вентиляционной матрице). Для этого потребуются переписать все определения микросхем на одном из промышленных языков описания

аппаратуры, а затем решить вопросы, связанные с реализацией ОЗУ, ПЗУ и устройств ввода-вывода на плате хоста. Один из таких пошаговых проектов, разработанный Майклом Шредером, приведен на сайте www.nand2tetris.org. Другой экстремальный подход — попытка эмуляции Nask, виртуальной машины или даже платформы Jack на существующем аппаратном устройстве, таком как мобильный телефон. Любой из этих проектов, по всей видимости, будет подразумевать уменьшение экрана для сохранения разумной стоимости аппаратных ресурсов.

Улучшения аппаратной части

Хотя Nask является компьютером с *хранимыми программами*, программу, которую он выполняет, приходится предварительно сохранять в его устройстве ПЗУ (ROM). В нынешней архитектуре Nask нет возможности загружать в компьютер другую программу, за исключением имитации замены всей физической микросхемы ПЗУ.

Добавление возможности загрузки программ сбалансированным способом, вероятно, потребует изменений на нескольких уровнях иерархии. Аппаратное обеспечение Nask нужно будет модифицировать так, чтобы загруженные программы находились в записываемом ОЗУ (RAM), а не в ПЗУ. Также, по всей видимости, для сохранения программ следует добавить в аппаратное обеспечение какой-то тип постоянного хранилища, например, встроенную микросхему массового хранения данных. Для работы с этим постоянным устройством хранения данных, а также с новой логикой загрузки и запуска программ придется расширить операционную систему. На этом этапе пригодится *оболочка* пользовательского интерфейса ОС, предлагающая команды управления файлами и программами.

Высокоуровневые языки

Как и все профессионалы, программисты испытывают сильные чувства к своим инструментам, языкам программирования, и любят персонализировать их, настраивая по своим предпочтениям. И действительно,

язык Jack оставляет желать лучшего и его можно в значительной степени улучшить. Некоторые изменения будут просты, другие сложнее, а некоторые — например, добавление наследования — скорее всего, потребуют изменения спецификации VM. Другой вариант — реализация на платформе Hack более высокоуровневых языков. Как, например, насчет реализации Scheme?

Оптимизация

Наше путешествие «От Nand до “Тетриса”» почти полностью обошло стороной вопросы оптимизации (за исключением операционной системы, в которой были предусмотрены некоторые меры по повышению эффективности). Оптимизация — это отличное игровое поле для хакеров. Можно начать с локальных оптимизаций в аппаратном обеспечении или в компиляторе, но наилучшие результаты получатся при оптимизации транслятора виртуальной машины. Например, можно уменьшить размер и повысить эффективность генерируемого ассемблерного кода. Оптимизации более глобального масштаба будут связаны с изменением спецификаций машинного языка или языка виртуальной машины.

Обмен данными

Разве не было бы здорово подключить компьютер Hack к Интернету? Это можно сделать, добавив в аппаратное обеспечение встроенную микросхему обмена данными и написав класс ОС для работы с ней и для обработки протоколов связи более высокого уровня. Тогда для некоторых других программ нужно будет предусмотреть средства общения с этим встроенным чипом и разработать интерфейс для подключения к Интернету. Вполне осуществимым и достойным проектом представляется проект по созданию веб-браузера Jack, понимающего протокол HTTP.

Это лишь некоторые из наших дизайнерских идей, к воплощению которых нам не терпится перейти, а какие идеи пришли в голову вам?

Приложение 1. Построение булевых функций

Благодаря логике мы доказываем, благодаря интуиции открываем.

— Анри Пуанкаре, (1854–1912)

В главе 1 мы сделали следующие утверждения, не приводя их доказательств.

- При наличии таблицы истинности некоей булевой функции можно построить булево выражение, соответствующее этой функции.
- Любую булеву функцию можно выразить с помощью только операторов And, Or и Not (И, ИЛИ и НЕ).
- Любую булеву функцию можно выразить с помощью только оператора Nand (И-НЕ).

В данном приложении приведены доказательства этих утверждений и показана их взаимосвязь. Кроме того, в приложении показан процесс упрощения булевых выражений с помощью булевой алгебры.

П1.1. Булева алгебра

Булевы операторы And, Or и Not (И, ИЛИ и НЕ) обладают полезными алгебраическими свойствами. Мы вкратце опишем некоторые из этих свойств, отметив, что их доказательства можно легко получить

из соответствующих таблиц истинности, показанных на иллюстрации 1.1 в главе 1.

Коммутативные законы:	$x \text{ And } y = y \text{ And } x$ $x \text{ Or } y = y \text{ Or } x$
Законы ассоциативности:	$x \text{ And } (y \text{ And } z) = (x \text{ And } y) \text{ And } z$ $x \text{ Or } (y \text{ Or } z) = (x \text{ Or } y) \text{ Or } z$
Законы дистрибутивности:	$x \text{ And } (y \text{ Or } z) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$ $x \text{ Or } (y \text{ And } z) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$
Законы де Моргана:	$\text{Not}(x \text{ And } y) = \text{Not}(x) \text{ Or } \text{Not}(y)$ $\text{Not}(x \text{ Or } y) = \text{Not}(x) \text{ And } \text{Not}(y)$
Законы идемпотентности:	$x \text{ And } x = x$ $x \text{ Or } x = x$

Эти алгебраические законы можно использовать для упрощения булевых функций. Например, рассмотрим функцию $\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y))$. Можно ли свести ее к более простой форме? Давайте попробуем и посмотрим, что у нас получится:

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y)) =$	// по закону де Моргана:
$\text{Not}(\text{Not}(x) \text{ And } (\text{Not}(x) \text{ And } \text{Not}(y))) =$	// по закону ассоциативности:
$\text{Not}((\text{Not}(x) \text{ And } \text{Not}(x)) \text{ And } \text{Not}(y)) =$	// по закону идемпотентности:
$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(y)) =$	// по закону де Моргана:
$\text{Not}(\text{Not}(x)) \text{ Or } \text{Not}(\text{Not}(y)) =$	// по двойному отрицанию:
$x \text{ Or } y$	

Упрощения булевых выражений, подобные только что проиллюстрированному, имеют значительные практические последствия.

Например, изначальное булево выражение $\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y))$ аппаратно можно было бы реализовать с помощью пяти логических вентилях, тогда как упрощенное выражение $x \text{ Or } y$ можно реализовать с помощью всего лишь одного логического вентиля. Оба выражения подразумевают одинаковую функциональность, но последнее в пять раз эффективнее с точки зрения стоимости, энергии и скорости вычислений.

Сведение булевых выражений к более простым — это искусство, требующее опыта и проницательности. Существуют различные инструменты и методы сокращения, но задача остается сложной. В целом сведение булева выражения к его простейшей форме является *NP-трудной* задачей.

П1.2. Построение булевых функций

Допустим, у нас имеется таблица истинности некоей булевой функции. Как на основе этой таблицы построить булево выражение, представляющее эту функцию? И можно ли с уверенностью утверждать, что каждую булеву функцию, заданную таблицей истинности, можно также задать булевым выражением?

На эти вопросы есть вполне удовлетворительные ответы. Во-первых, да: каждую булеву функцию можно представить булевым выражением. Более того, для этого существует определенный алгоритм. Для примера обратимся к иллюстрации П1.1 и сосредоточимся на ее крайних левых четырех столбцах. Эти столбцы задают таблицу истинности некоторой функции $f(x, y, z)$ с тремя переменными. Наша цель — построить на основе этих данных булево выражение, задающее эту функцию.

x	y	z	$f(x,y,z)$	$f_3(x,y,z)$	$f_5(x,y,z)$	$f_7(x,y,z)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
1	0	0	1	0	1	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
1	1	1	0	0	0	0

$f_3(x,y,z) = \text{Not}(x) \text{ And } y \text{ And Not}(z)$
 $f_5(x,y,z) = x \text{ And Not}(y) \text{ And Not}(z)$
 $f_7(x,y,z) = x \text{ And } y \text{ And Not}(z)$
 $f(x,y,z) = f_3(x,y,z) \text{ Or } f_5(x,y,z) \text{ Or } f_7(x,y,z)$

Иллюстрация П1.1. Построение булевой функции на основе таблицы истинности (пример).

Опишем на данном конкретном примере пошаговый алгоритм синтеза. Начнем с того, что сосредоточимся только на тех строках таблицы истинности, в которых значение функции равно 1. Для функции на иллюстрации П1.1 это происходит в строках 3, 5 и 7. Для каждой такой строки i определим булеву функцию f_i , возвращающую 0 для всех значений переменных, кроме значений переменных в строке i , для которых функция возвращает 1. Таблица истинности на рисунке П1.1 дает три такие функции, определенные в трех крайних правых столбцах таблицы. Каждую из таких функций f_i можно представить в виде конъюнкции (связывания оператором And) трех членов, по одному на каждую переменную x , y и z , выраженных либо в виде себя, либо своего отрицания, в зависимости от их значений 1 или 0 в строке i . Этот метод дает нам три функции f_3 , f_5 и f_7 , перечисленные в нижней части таблицы. Поскольку эти функции описывают единственные случаи, когда булева функция f принимает значение 1, можно утверждать, что функция f может быть представлена булевым выражением $f(x, y, z) = f_3(x, y, z) \text{ Or } f_5(x, y, z) \text{ Or } f_7(x, y, z)$. Разложим его: $f(x, y, z) = (\text{Not } (x) \text{ And } y \text{ And Not } (z)) \text{ Or } (x \text{ And Not } (y) \text{ And Not } (z)) \text{ Or } (x \text{ And } y \text{ And Not } (z))$.

Если не вдаваться в утомительные формальности, то на этом примере видно, что любую булеву функцию можно задать в виде булева выражения с очень специфической структурой: это дизъюнкция (Or) всех конъюнктивных (And) функций f_i , построение которых было только что описано. Это выражение, являющееся своего рода булевой версией суммы произведений, иногда называют *дизъюнктивной нормальной формой* (ДНФ) функции.

Обратите внимание, что если функция имеет много переменных и, следовательно, таблица истинности имеет экспоненциально много строк, то ДНФ может получиться очень длинной и громоздкой. На этом этапе преобразовать выражение в более эффективный и пригодный для дальнейшей работы вид помогут булева алгебра и различные методы сокращения.

П1.3. Выразительная сила Nand

Как следует из названия нашего общего проекта «От Nand до “Тетриса”», любой компьютер можно построить, используя только вентили Nand. Доказать это можно двумя способами. Во-первых, на самом деле построить компьютер только из вентилях Nand, чем мы и занимаемся в первой части книги. Другой способ — предоставить формальное доказательство, что мы сделаем далее.

Лемма 1: любую булеву функцию можно представить в виде булева выражения, содержащего только операторы And, Or и Not.

Доказательство: для любой булевой функции можно построить соответствующую ей таблицу истинности. И, как только что мы показали, на основе любой таблицы истинности можно построить формулу в дизъюнктивной нормальной форме, отображающую дизъюнкцию (Or) всех конъюнктивных (And) операций над переменными или их отрицаниями. Иными словами, это выражение, содержащее только операторы And, Or и Not.

Чтобы оценить значение данного результата, рассмотрим бесконечное число функций, которые могут быть определены над *целыми числами* (а не только двоичными). Было бы неплохо, если бы каждую такую функцию можно было представить в виде алгебраического выражения, в котором фигурировали бы только сложение, умножение и отрицание. Оказывается, подавляющее большинство функций над целыми числами, например $f(x) = 2x$ для $x \neq 7$ и $f(7) = 312$, не могут быть выражены с помощью замкнутой алгебраической формы. Но что касается мира двоичных чисел, то в нем, благодаря конечному числу значений, которые может принимать каждая переменная (0 или 1), любую булеву функцию можно выразить с помощью всего лишь трех операторов And, Or и Not. Практическое значение этого замечательного свойства огромно: любой компьютер можно построить на основе всего лишь трех логических элементов: And, Or и Not.

Но можно ли добиться еще большего упрощения?

Лемма 2: любую булеву функцию можно представить в виде булева выражения, содержащего только операторы Not и And.

Доказательство: согласно закону де Моргана, оператор Or можно выразить с помощью операторов Not и And. Объединив это с результатом Леммы 1, получаем доказательство Леммы 2. Итак, мы еще больше сократили количество операторов.

Но что дальше?

Теорема: любую булеву функцию можно представить в виде булева выражения, содержащего только операторы Nand.

Доказательство: При анализе таблицы истинности для оператора Nand (предпоследний ряд на иллюстрации 1.2 в главе 1) можно сформулировать два его свойства:

$$\text{— } \text{Not}(x) = \text{Nand}(x, x)$$

Иными словами: если переменные x и y функции Nand принимают одно и то же значение (0 или 1), функция эквивалентна отрицанию этого значения.

$$\text{— } \text{And}(x, y) = \text{Not}(\text{Nand}(x, y))$$

Легко доказать, что таблицы истинности для обеих частей уравнения идентичны. И мы только что показали, что функцию Not можно выразить через функцию Nand.

Объединив оба результата с результатом Леммы 2, мы получаем доказательство того, что любую булеву функцию можно представить в виде булева выражения, содержащего одни лишь операторы Nand.

Этот замечательный результат, который вполне можно назвать фундаментальной теоремой логического проектирования, гласит, что компьютеры можно построить на основе только одного элементарного логического элемента: логических вентилях, реализующих функцию Nand. Другими словами, если в нашем распоряжении будет столько вентилях Nand, сколько потребуется, то мы их можем соединить

в схемы, реализующие любую заданную булеву функцию: все, что нам нужно сделать, это придумать правильную схему.

И действительно, большинство современных компьютеров основаны на аппаратной инфраструктуре, состоящей из миллиардов вентилях Nand (или Nor, обладающих аналогичными свойствами). Впрочем, на практике вовсе не обязательно ограничиваться одними лишь Nand-вентильями. Если инженеры-электрики и физики смогут придумать эффективные и недорогие физические реализации других элементарных логических вентилях, мы с удовольствием будем использовать их в качестве непосредственных элементарных строительных блоков. Это прагматическое замечание нисколько не отрицает важности доказанной выше теоремы.

Приложение 2. Язык описания аппаратуры

Интеллект — это способность создавать искусственные объекты, особенно инструменты для создания инструментов.

— Анри Бергсон (1859–1941)

Это приложение состоит из двух основных частей. Разделы П2.1–П2.5 описывают язык HDL, используемый в книге и в проектах. Раздел П2.6 под названием «Практический справочник по HDL» содержит набор важных советов для успешного выполнения проектов по аппаратному обеспечению.

Язык описания аппаратуры (HDL) представляет собой формальное средство описания *микросхем* (в разговорном языке называемых также *чипами*): объектов, *интерфейсы* которых состоят из входных и выходных *контактов*, передающих двоичные сигналы; эти микросхемы реализуются на основе соединенных между собой других, более низкоразрядных микросхем. В данном приложении описывается язык HDL, используемый в практическом курсе «От Nand до “Тетриса”». Основные сведения об этом языке приведены в главе 1 (в частности, в разделе 1.3), и с ними нужно ознакомиться до того, как переходить к приложению.

П2.1. Основы HDL

Язык HDL, используемый в практическом курсе «От Nand до “Тетриса”», довольно прост, и лучший способ изучить его — это поиграть

с программами HDL с помощью прилагаемого симулятора аппаратуры. Мы советуем вам как можно раньше приступить к экспериментам, начиная со следующего примера.

Пример: предположим, что нам нужно проверить, имеют ли три 1-битные переменные a , b и c одно и то же значение. Один из способов проверить это — оценить булеву функцию $((a \neq b) \vee (b \neq c))$. С учетом того, что двоичный оператор «не равно» можно реализовать с помощью вентиля Хор, данную функцию можно реализовать с помощью HDL-программы, показанной на иллюстрации П2.1.

В реализации Eq3.hdl используются четыре *микросхемы-части*: два вентиля Хор, один вентиль Ор и один вентиль Not. Чтобы реализовать логику, выраженную в выражении $((a \neq b) \vee (b \neq c))$, HDL-программист соединяет между собой микросхемы-части, создавая и именую три внутренних вывода: neq1, neq2 и outOr.

Интерфейс	{	/** Если три данных бита равны, на out выводится 1; иначе на out выводится 0. */
		CHIP Eq3 {
		IN a, b, c;
		OUT out;
		PARTS:
Реализация	{	Xor(a\a, b\b, out\neq1); // Xor(a,b) → neq1
		Xor(a\b, b\c, out\neq2); // Xor(b,c) → neq2
		Or (a\neq1, b\neq2, out\outOr); // Or(neq1,neq2) → outOr
		Not(in\outOr, out\out); // Not(outOr) → out
		}

Иллюстрация П2.1. Пример HDL-программы.

В отличие от внутренних контактов, которые можно создавать и называть по своему усмотрению, программист на языке HDL не контролирует имена входных и выходных выводов. Обычно их предоставляют разработчики микросхем, и они документируются в соответствующих API. Мы в нашем практическом курсе «От Nand до “Тетриса”» для всех микросхем, которые вам предлагается реализовать, предоставляем *файлы-заглушки*. Каждый файл-заглушка содержит интерфейс микросхемы, но в нем отсутствует реализация. Контракт заключается

в следующем: ниже высказывания PARTS вам разрешается делать все, что вы хотите; выше высказывания PARTS ничего менять нельзя.

В примере с Eq3 так получилось, что первые два входа микросхемы Eq3 и два входа ее микросхем-частей Хог и Ог имеют одинаковые имена (a и b). Аналогично выход микросхемы Eq3 и выход части Not имеют одинаковые имена (out). В результате получаются такие связки, как $a = a$, $b = b$ и $out = out$. Такие связки могут выглядеть необычно, но они часто встречаются в HDL-программах, и к ним привыкаешь. Позже в приложении мы приведем простое правило, которое проясняет смысл этих привязок.

Важно отметить, что программисту не нужно беспокоиться о реализации составных частей микросхемы. Элементы микросхемы используются как абстракции типа «черный ящик», позволяя программисту сосредоточиться только на том, как их лучше всего расположить, чтобы реализовать функцию микросхемы. Благодаря такой модульности HDL-программы бывают короткими, легко читаемыми и поддающимися модульному тестированию.

Микросхемы, разработанные на базе HDL, такие как Eq3.hdl, можно тестировать с помощью компьютерной программы, называемой *симулятором аппаратуры*. Когда мы даем симулятору команду оценить данную микросхему, симулятор оценивает все части микросхемы, указанные в разделе PARTS. Это, в свою очередь, требует оценки их составных деталей более низкого уровня и т. д. Такая нисходящая рекурсивная проверка может привести к созданию все более ветвящейся и расширяющейся огромной иерархии вплоть до конечных вентилей Nand, из которых сделаны все микросхемы. Такого ресурсозатратного спуска можно избежать, используя *встроенные микросхемы*, как мы вскоре объясним.

HDL — это декларативный язык: HDL-программы можно рассматривать как текстовые спецификации диаграмм микросхем. Для каждой микросхемы *chipName*, которая появляется на диаграмме, программист записывает высказывание *chipName (...)* в разделе PARTS HDL-программы. Поскольку язык предназначен для описания связей, а не процессов, порядок высказываний в разделе PARTS несущественен: если

части микросхемы соединены между собой правильно, микросхема будет функционировать, как заявлено. То, что высказывания в языке HDL могут идти друг за другом в любом порядке без влияния на поведение микросхемы, может показаться странным для читателей, привыкших к обычному программированию. Помните: HDL — это не язык программирования, это язык спецификации.

Пробелы, комментарии, условности регистра. HDL чувствителен к регистру: foo и Foo представляют собой два разных объекта. Ключевые слова HDL записываются заглавными буквами. Пробелы, символы новой строки и комментарии игнорируются. Поддерживаются следующие форматы комментариев:

```
// Комментарий до конца строки  
/* Комментарий до закрытия */  
/** Комментарий к документации API */
```

Контакты. В HDL-программах описываются три типа контактов: входные контакты, выходные контакты и внутренние контакты. Последние служат для соединения выходов одних частей со входами других частей. По умолчанию предполагается, что контакты передают однобитовые значения 0 или 1. Также можно объявлять и использовать многобитные шинные выводы, как описано далее в этом приложении.

Имена микросхем и контактов представляют собой любую последовательность букв и цифр, не начинающуюся с цифры (некоторые симуляторы аппаратуры не позволяют использовать дефисы). По соглашению имена микросхем начинаются с заглавной буквы, а имена контактов — со строчной. Для удобства чтения имена могут включать заглавные буквы, например, xorResult. HDL-программы хранятся в файлах .hdl. Имя микросхемы, объявленной в HDL-высказывании CHIP Xxx, должно совпадать с префиксом имени файла Xxx.hdl.

Структура программы: HDL-программа состоит из интерфейса и реализации. Интерфейс состоит из документации API микросхемы,


```
chipPart2(..., in=v, ...);    //входной контакт in части
                             chipPart2 соединен с v.
chipPart3(..., in1=v, ..., in2=v, ...); //входные контакты
                                     in1 и in2 части
                                     chipPart3 также
                                     соединены с v.
```

У каждого контакта может быть только один вход, но выход может соединяться с несколькими входами одной или нескольких микросхем-частей. В приведенном выше примере выход контакта *v* может передавать данные на три разных входа. Это HDL-эквивалент *разветвлений* в схемах микросхем.

Значение *a = a*: многие микросхемы в платформе Nask используют одинаковые имена выводов. Как показано на иллюстрации П2.1, это приводит к высказываниям типа *Xor(a = a, b = b, out = neq1)*. Первые два соединения говорят о том, что данные со входов *a* и *b* реализуемой микросхемы (*Eq3*) подаются на входы *a* и *b* микросхемы-части *Xor*. Третье соединение передает данные с выхода *out* микросхемы-части *Xor* на внутренний контакт *neq1*. Во всем этом помогает разобраться простое правило: в каждом операторе микросхемы-части левая часть каждой связки = всегда обозначает вход или выход этой *микросхемы-части*, а правая часть всегда обозначает вход, выход или внутренний контакт *реализуемой микросхемы*.

П2.2. Многобитные шины

Каждый вход, выход или внутренний контакт в HDL-программе может передавать либо однобитное значение (по умолчанию), либо многобитное значение, и в таком случае этот контакт называется *шиной*.

Нумерация битов и синтаксис шины: биты нумеруются справа налево, начиная с 0. Например, *sel = 110* означает, что *sel[2] = 1*, *sel[1] = 1* и *sel[0] = 0*.

Входные и выходные контакты шины: ширина битов этих контактов задается при их объявлении в операторах IN и OUT микросхемы. Синтаксис имеет вид $x[n]$, где x и n определяют имя контакта и ширину в битах соответственно.

Внутренние контакты шины: ширина внутренних контактов выводится неявно из высказываний, в которых они объявляются, следующим образом:

```
chipPart1 (... , x[i] = u, ... );
chipPart2 (... , x[i..j] = v, ... );
```

где x — вход или выход микросхемы-части. Первое высказывание определяет u как однобитный внутренний контакт и устанавливает его значение равным $x[i]$. Второе высказывание определяет v как внутреннюю шину шириной $j - i + 1$ бит и устанавливает значение ее битов с индексами от i до j (включительно).

В отличие от входных и выходных контактов, внутренние контакты (например, u и v) не могут иметь индексов. Например, выражение $u[i]$ не допускается.

```
// Устанавливает значение
// out=Not(in) побитово
CHIP Not8 {
    IN  in[8];
    OUT out[8];
    ...
}

CHIP Foo {
    ...
    PARTS
    ...
    Not8(in[0..1] \ true,
          in[3..5] \ six,
          in[7]    \ true,
          out[3..7] \ out1,
          ...
    );
    ...
}
```

Допущение: *six* — внутренняя шина, передающая значение **110**.
out1 — внутренний контакт, созданный высказыванием для микросхемы-части *Not8*.

Ниже: итоговые значения на входе *in* микросхемы *Not8* и на контакте *out1*.

	7	6	5	4	3	2	1	0
in:	1	0	1	1	0	0	1	1

	4	3	2	1	0
out1:	0	1	0	0	1

Иллюстрация П2.2. Шины в действии (пример).

Шины true/false: для определения шин можно также использовать константы true (1) и false (0). Предположим, например, что *x* — это 8-битная шина-контакт, и рассмотрим следующее высказывание:

```
chipPart (... , x[0..2]=true, ..., x[6..7]=true, ...).
```

Это высказывание присваивает *x* значение 11000111. Обратите внимание, что значение неупомянутых битов по умолчанию равно false (0). На иллюстрации П2.2 приведен еще один пример.

П2.3. Встроенные микросхемы

Микросхемы могут иметь либо *собственную (нативную)* реализацию, написанную на языке HDL, либо *встроенную* реализацию в виде готового исполняемого модуля, написанного на высокоуровневом языке программирования. Поскольку симулятор аппаратуры «От Nand до “Тетриса”» был написан на языке Java, то было удобно реализовать встроенные микросхемы как классы Java. Так что прежде чем создавать, скажем, микросхему Mux на языке HDL, пользователь может загрузить в симулятор аппаратуры встроенную микросхему Mux и поэкспериментировать с ней. Поведение встроенного чипа Mux обеспечивается файлом класса Java с именем Mux.class, входящим в состав программного обеспечения симулятора.

Компьютер Nask состоит примерно из тридцати микросхем, перечисленных в приложении 4. Две из этих микросхем, Nand и DFF, считаются заданными, элементарными, или *примитивными*, подобно аксиомам в логике. Симулятор аппаратуры реализует эти микросхемы, вызывая их встроенные реализации. Как следствие, в проектах «От Nand до “Тетриса”» микросхемами Nand и DFF можно пользоваться без создания их средствами HDL.

Проекты 1, 2, 3 и 5 посвящены созданию HDL-реализаций остальных чипов, перечисленных в приложении 4. Все эти микросхемы, за исключением микросхем CPU и Computer, также имеют встроенные реализации. Как объяснялось в главе 1, это было сделано для того, чтобы облегчить моделирование их поведения.

Встроенные микросхемы — библиотека примерно из тридцати файлов вида *chipName.class* — поставляются вместе с симулятором аппаратуры и содержатся в папке `nand2tetris/tools/builtInChips` на вашем компьютере. Интерфейсы встроенных микросхем идентичны интерфейсам обычных HDL-микросхем. Поэтому для каждого файла `.class` имеется соответствующий файл `.hdl`, обеспечивающий интерфейс встроенной микросхемы. На иллюстрации П2.3 показано типичное HDL-определение встроенной микросхемы.

```
/** 16-битный вентиль And, реализованный как встроенный чип. */  
CHIP And16 {  
    IN  a[16], b[16];  
    OUT out[16];  
    BUILTIN And16;  
}
```

Реализовано с помощью
`tools/builtInChips/And16.class`

Иллюстрация П2.3. Пример определения встроенной микросхемы.

Важно помнить, что прилагаемый симулятор аппаратуры — это инструмент общего назначения, в то время как компьютер Hack, создаваемый в рамках проекта «От Nand до “Тетриса”», — конкретная аппаратная платформа. Симулятор аппаратуры можно использовать для создания вентилях, микросхем и платформ, не имеющих отношения к Hack. Поэтому, обсуждая понятие встроенных микросхем, полезно расширить перспективу и описать их общую пользу для поддержки любого возможного проекта построения аппаратного обеспечения. В целом встроенные микросхемы предоставляют следующие услуги.

Основание: встроенные микросхемы могут служить реализациями микросхем, которые считаются *заданными*, или *примитивными*. Например, в компьютере Hack заданные микросхемы — это Nand и DFF.

Эффективность: некоторые микросхемы, например блоки оперативной памяти, состоят из множества микросхем более низкого уровня. Когда мы используем такие микросхемы в качестве частей микросхемы более высокого уровня, симулятор аппаратуры должен также

смоделировать и их поведение. Это делается посредством рекурсивной проверки всех микросхем нижнего уровня, из которых они состоят. В результате симуляция может замедлиться, а ее эффективность понизиться. Использование встроенных микросхем-частей вместо обычных микросхем на базе HDL значительно ускоряет моделирование.

Тестирование микросхем: HDL-программы используют микросхемы-части как абстракции, без учета их реализации. Поэтому при создании новой микросхемы всегда рекомендуется использовать встроенные микросхемы-части. Такая практика повышает эффективность и минимизирует ошибки.

Визуализация: если разработчик хочет позволить пользователям «посмотреть», как работают микросхемы, и по возможности интерактивно изменять внутреннее состояние моделируемой микросхемы, он может предоставить ее встроенную реализацию, имеющую графический интерфейс пользователя. Этот графический интерфейс будет отображаться всякий раз при загрузке в симулятор встроенной микросхемы или при ее вызове как части моделируемой микросхемы. За исключением этих побочных визуальных эффектов, микросхемы с графическим интерфейсом ведут себя точно так же, как и любые другие, и их точно так же можно использовать. Подробнее о микросхемах с графическим интерфейсом говорится в разделе П2.5.

Расширение: если вы хотите реализовать новое устройство ввода/вывода или вообще создать новую аппаратную платформу (отличную от Nask), то эти конструкции можно создавать при помощи встроенных микросхем. Более подробная информация о разработке дополнительных или новых функциональных возможностей содержится в главе 13.

П2.4. Последовательностные микросхемы

Микросхемы бывают *комбинационными* или *последовательностными*. Комбинационные микросхемы не зависят от времени: они реагируют

на изменения своих входов мгновенно. Последовательностные микросхемы (называемые также *синхронизируемыми* или *тактируемыми*) зависят от времени: когда пользователь или сценарий тестирования меняют значения на входах последовательной микросхемы, значения на выходах микросхемы меняются только в начале следующей *единицы времени*, также называемой *циклом*. Симулятор аппаратуры моделирует ход времени с помощью симуляции генератора тактовой частоты (*clock*).

Генератор тактовой частоты (*clock*): двухфазный генератор тактовой частоты симулятора испускает бесконечную последовательность значений, обозначаемых 0, 0+, 1, 1+, 2, 2+, 3, 3+ и т. д. Прогрессия этого дискретного временного ряда контролируется двумя командами симулятора, называемыми *tick* и *tock* (буквально «тик» и «так»). *Tick* меняет значение *clock* с *t* на *t+*, а *tock* — с *t+* на *t+1*, в результате чего происходит переход к следующей единице времени. Реальное время, прошедшее за этот период, не имеет значения для целей моделирования, поскольку симулируемое время управляется пользователем или тестовым скриптом следующим образом.

Во-первых, всякий раз, когда в симулятор загружается последовательностная микросхема, графический интерфейс пользователя активирует кнопку в форме часов (при моделировании комбинационных микросхем она неактивна и затемнена). Одно нажатие на эту кнопку (*tick*) завершает первую фазу тактового цикла, а последующее нажатие (*tock*) завершает вторую фазу цикла, начиная первую фазу следующего цикла, и т. д.

В качестве альтернативы можно запустить генератор тактовой частоты из тестового сценария. Например, последовательность команд сценария `repeat n {tick, tock, output}` предписывает симулятору выполнить *n* циклов тактовой частоты и вывести в процессе работы некоторые значения. В приложении 3 содержится документация *Языка описания тестов* (TDL) с описанием этих команд.

Двухфазные единицы времени, генерируемые «часами», регулируют работу всех последовательностных микросхем-частей в реализуемой микросхеме. Во время первой фазы единицы времени (*tick*)

значения на входах каждой последовательностной микросхемы-части влияют на внутреннее состояние этой микросхемы в соответствии с ее логикой. Во время второй фазы единицы времени (*tock*) на выходах микросхемы устанавливаются новые значения. Таким образом, если взглянуть на последовательностную микросхему как бы «со стороны», то можно увидеть, как значения на ее выходах стабилизируются только во время фазы *tock* — в точке перехода между двумя последовательными единицами времени.

Повторим еще раз: комбинационные микросхемы совершенно не обращают внимания на генератор тактовой частоты. В проектах «От Nand до “Тетриса”» все логические вентили и микросхемы, построенные в главах 1–2 вплоть до АЛУ, комбинационные. Все регистры и блоки памяти, построенные в главе 3, последовательностные. По умолчанию микросхемы являются комбинационными; микросхема может стать *последовательностной* явно или неявно, как показано ниже.

Последовательностные встроенные микросхемы: для *встроенной микросхемы* зависимость от генератора тактовой частоты может быть явно объявлена высказыванием типа:

```
CLOCKED pin, pin, ..., pin;
```

где каждый *pin* — это один из входных или выходных контактов микросхемы. Если в список *CLOCKED* включен входной контакт *x*, то это означает, что изменение значения на входе *x* должно повлиять на выходы микросхемы только в начале следующей единицы времени. Если в список *CLOCKED* включен выходной контакт *x*, то это означает, что изменения значений на любом из входов микросхемы должны повлиять на *x* только в начале следующей единицы времени. На иллюстрации П2.4 представлено определение самой основной, встроенной, последовательностной микросхемы платформы Nask — DFF.

Можно объявить тактируемыми только некоторые из входных или выходных контактов микросхемы. В этом случае изменения значений на нетактируемых входах влияют на нетактируемые выходы мгновенно. Именно так реализованы выводы *address* в блоках оперативной

памяти: логика адресации является комбинационной и не зависит от генератора тактовой частоты.

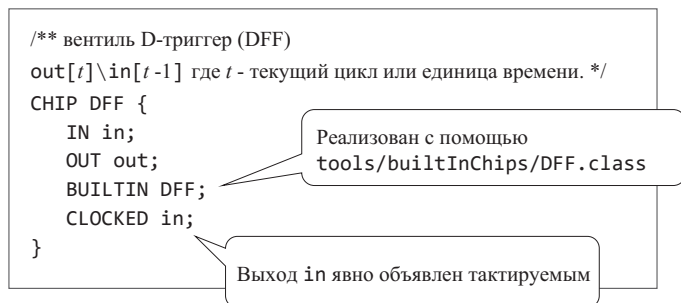


Иллюстрация П2.4. Определение DFF.

Можно также объявить ключевое слово `CLOCKED` с пустым списком контактов. Такое заявление подразумевает, что микросхема может менять свое внутреннее состояние в зависимости от тактового генератора, но ее поведение на входе-выходе будет комбинационным, не зависящим от генератора тактовой частоты.

Последовательностные составные микросхемы: свойство `CLOCKED` может задаваться явно только для встроенных микросхем. Как же тогда симулятор узнает, что данная микросхема-часть является последовательностной? Если микросхема не встроенная, то она считается тактируемой, только если в ее состав входит одна или несколько тактируемых частей. Свойство тактирования проверяется рекурсивно, вниз по всей иерархии микросхем до явно тактируемой встроенной микросхемы. Если такая микросхема найдена, она делает «тактируемыми» все микросхемы, которые зависят от нее (находятся выше нее по иерархии). Поэтому в компьютере Nask все микросхемы, включающие в свой состав одну или несколько микросхем DFF, явно или косвенно считаются тактируемыми.

Отсюда следует, что если микросхема не встроенная, то из ее HDL-кода невозможно определить, является ли она последовательностной или нет. *Лучший практический совет:* архитектор микросхемы должен предоставлять эту информацию в документации API микросхемы.

Контуры обратной связи: если на вход микросхемы подается один из выходов этой же микросхемы — либо напрямую, либо через (возможно, длинный) путь зависимостей, — мы говорим, что микросхема содержит *контур обратной связи*. Рассмотрим, например, следующие два высказывания микросхемы-части.

```
Not (in=loop1, out=loop1) // Неправильная петля
                           обратной связи
DFF (in=loop2, out=loop2) // Правильная петля обратной
                           связи
```

В обоих примерах внутренний выход (loop1 или loop2) пытается подать сигнал на вход той же микросхемы, создавая контур обратной связи. Разница между двумя примерами заключается в том, что Not — это комбинационная микросхема, в то время как DFF — последовательностная. В примере с Not loop1 создает мгновенную и неконтролируемую зависимость между выходом и входом — состояние, которое иногда называют «гонкой данных». Напротив, в случае с DFF зависимость между выходом и входом, которую создает контакт loop2, определяется генератором тактовой частоты, поскольку вход in чипа DFF объявлен тактируемым. Поэтому out(t) — это скорее не функция in(t), а функция in(t-1).

Оценивая микросхему, симулятор рекурсивно проверяет, не создают ли ее различные соединения контуры обратной связи. Для каждого контура он проверяет, проходит ли контур через какой-либо тактируемый контакт. Если это верно, то контур считается допустимым. В противном случае симулятор прекращает обработку и выдает сообщение об ошибке. Это делается для предотвращения неконтролируемых «гонок данных».

П2.5. Визуализация микросхем

Встроенные микросхемы могут дополняться *графическим интерфейсом (GUI)*. Эти микросхемы имеют визуальные побочные эффекты,

предназначенные для анимации некоторых операций микросхемы. Оценивая микросхему-часть с GUI, симулятор выводит на экран некое графическое изображение. С помощью этого изображения, которое может включать в себя интерактивные элементы, пользователь может проверить текущее состояние микросхемы или изменить его. Допустимые действия для микросхем с GUI определяются разработчиком реализации встроенной микросхемы.

В текущей версии симулятора аппаратуры имеются следующие встроенные микросхемы с поддержкой GUI.

ALU: отображает значения на входах и выходах АЛУ компьютера Hack и вычисляемую им в данный момент функцию.

Регистры (ARegister, DRegister, PC): отображает содержимое регистра, которое может поменять пользователь.

Микросхемы RAM (оперативной памяти): отображает прокручиваемую, похожую на массив таблицу, показывающую содержимое всех ячеек памяти, содержимое которых может изменять пользователь. Если содержимое ячейки памяти изменяется во время моделирования, соответствующая запись в графическом интерфейсе также изменяется.

Микросхема ПЗУ (ROM32K): то же изображение в виде таблицы-массива, что и у микросхем ОЗУ, плюс значок, позволяющий загрузить программу на машинном языке из внешнего текстового файла. Микросхема ROM32K служит в качестве памяти команд компьютера Hack.

Микросхема Screen: отображает окно размером 256 строк на 512 столбцов, имитирующее физический экран. Если во время моделирования один или несколько битов *карты памяти экрана*, находящейся в ОЗУ, меняются, то меняются и соответствующие пиксели в графическом интерфейсе экрана. Этот цикл непрерывного обновления встроен в реализацию симулятора.

Микросхема Keyboard: отображает значок клавиатуры. Щелчок по этому значку подключает реальную клавиатуру вашего компьютера к моделируемой микросхеме. С этого момента каждая клавиша, нажатая на реальной клавиатуре, перехватывается моделируемой микросхемой, и двоичный код этой клавиши появляется в *карте памяти клавиатуры*, находящейся в ОЗУ. Если пользователь перемещает фокус мыши в другую область графического интерфейса симулятора, управление клавиатурой возвращается к реальному компьютеру.

На иллюстрации П2.5 показана микросхема, в которой используются три микросхемы-части с графическим интерфейсом. На иллюстрации П2.6 показано, как симулятор обрабатывает эту микросхему. Согласно логике микросхемы GUIDemo, значение на входе *in* подается по двум направлениям: на вход *address* (номера регистра) микросхемы-части RAM16K и на вход *address* (номера регистра) микросхемы Screen. Кроме того, согласно логике микросхемы, значение на выходах *out* трех ее частей подается на три «тупиковых» разъема *a*, *b* и *c*. Эти бессмысленные соединения предназначены только для одной цели: проиллюстрировать работу симулятора со встроенными микросхемами-частями с графическим интерфейсом.

```
//Демонстрация чипов с GUI
//Логика чипов бессмысленна и используется только для того,
//чтобы заставить симулятор отображать эффекты GUI
//встроенных частей микросхемы
CHIP GUIDemo {
    IN  in[16], load, address[15];
    OUT out[16];
    PARTS:
    RAM16K(in\in, load\load, address\address[0..13], out\a);
    Screen(in\in, load\load, address\address[0..12], out\b);
    Keyboard(out\c);
}
```

Иллюстрация П.2.5. Микросхема, активирующая микросхемы-части с GUI.

Обратите внимание на то, как внесенные пользователем изменения повлияли на экран (шаг 4). Горизонтальная черта на экране — это побочный визуальный эффект хранения значения -1 в ячейке памяти 5012.

Поскольку 16-разрядный двоичный код -1 равен 1111111111111111 , компьютер рисует 16 пикселей, начиная со столбца 320 строки 156 — экранных координат, связанных с адресом 5012 оперативной памяти. Сопоставление адресов памяти с экранными координатами (*строка, столбец*) описано в главе 4 (раздел 4.2.5).

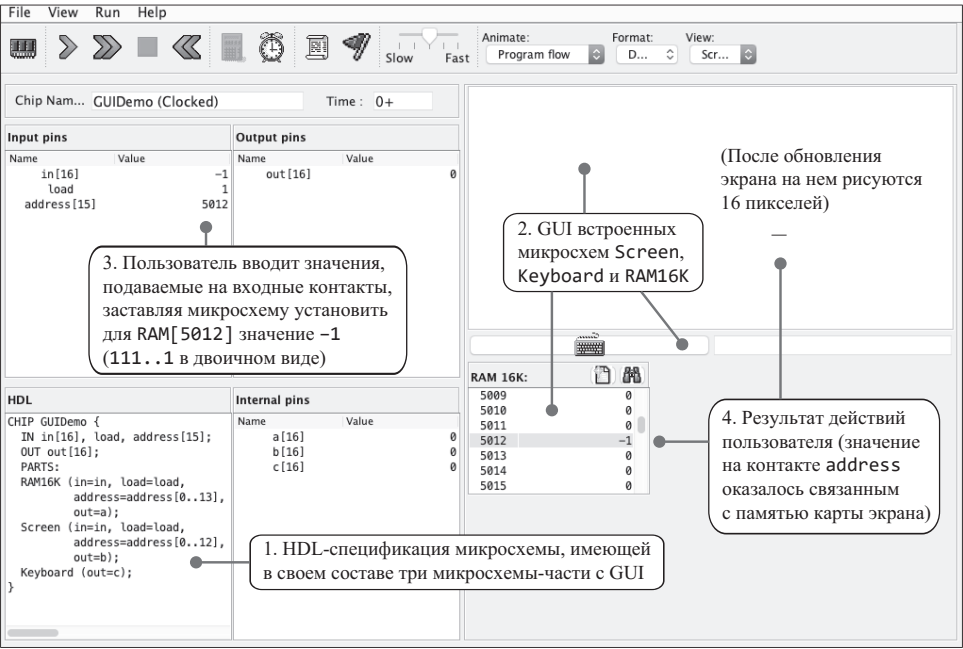


Иллюстрация П2.6. Демонстрация микросхем с GUI (графическими интерфейсами). Поскольку загруженная HDL-программа использует части микросхемы с графическим интерфейсом (шаг 1), симулятор отображает их соответствующие графические интерфейсы (шаг 2). Когда пользователь изменяет значения на входах микросхемы (шаг 3), симулятор отражает эти изменения в соответствующих графических интерфейсах (шаг 4).

П2.6. Практический справочник по HDL

В этом разделе приведены практические советы по разработке микросхем на языке HDL с использованием прилагаемого симулятора аппаратуры. Советы не расположены в каком-то определенном порядке.

Мы рекомендуем прочитать этот раздел один раз от начала до конца, а затем обращаться к нему по мере необходимости.

Микросхема («чип»): в папке `nand2tetris/projects` на вашем компьютере содержатся тринадцать подпапок, названных 01, 02... 13 (по номерам соответствующих глав). Папки аппаратных проектов — 01, 02, 03 и 05. Каждая папка аппаратного проекта содержит набор поставляемых *файлов-заглушек* HDL, по одному для каждой микросхемы, которую необходимо собрать. Поставляемые HDL-файлы не содержат реализаций; построение этих реализаций и есть цель проекта. Если вы будете собирать эти микросхемы не в том порядке, в котором они описаны в книге, то можете столкнуться с некоторыми трудностями. Предположим, например, что вы начинаете проект 1 со сборки микросхемы Xor. Если ваша реализация `Xor.hdl` включает в себя, допустим, микросхемы-части And и Or, но вы еще не реализовали `And.hdl` и `Or.hdl`, ваша программа `Xor.hdl` не будет работать, даже если ее реализация совершенно правильная.

Однако учтите, что, если в папке проекта не будет файлов `And.hdl` и `Or.hdl`, ваша программа `Xor.hdl` будет работать правильно. Симулятор аппаратуры, представляющий собой программу на языке Java, имеет встроенные реализации всех микросхем, необходимых для создания компьютера Hack (за исключением микросхем CPU и Computer). Оценивая очередную микросхему, скажем, And, он ищет файл `And.hdl` в текущей папке. Здесь возможны три варианта.

- HDL-файл не найден. В этом случае включается встроенная реализация микросхемы, которая заменяет отсутствующую HDL-реализацию.
- Найден файл-заглушка HDL. Симулятор попытается выполнить его. Если в нем не прописана реализация, выполнение завершается неудачей.
- Найден HDL-файл с HDL-реализацией. Симулятор выполняет его, сообщая об ошибках, если таковые имеются, в меру своих возможностей.

Лучший практический совет: вы можете следовать одному из двух планов. Во-первых, попробуйте создавать реализации микросхем в том порядке, в котором о них говорится в книге и в котором они перечисляются в описаниях проектов. Поскольку микросхемы рассматриваются «снизу вверх», от базовых к более сложным, в таком случае проблем с реализациями микросхем не будет — конечно, при условии, что вы правильно завершите реализацию каждой микросхемы, прежде чем переходить к реализации следующей.

Рекомендуемая альтернатива — создать подпапку, допустим, с именем `stubs`, и переместить в нее все поставляемые файлы-заглушки `.hdl`. Затем можно один за другим перемещать в рабочую папку файлы-заглушки, над которыми вы хотите поработать. Завершив успешно реализацию микросхемы, переместите ее файл в другую подпапку, допустим, с именем `completed`. При таком подходе симулятор будет всегда использовать встроенные микросхемы, потому что в рабочей папке находится только один файл `.hdl`, над которым вы работаете в данный момент (а также поставляемые файлы `.tst` и `.cmp`).

HDL-файлы и тестовые сценарии: файл `.hdl`, над которым вы работаете, и связанный с ним файл тестового сценария `.tst` должны находиться в одной папке. Каждый прилагаемый тестовый сценарий начинается с команды `load`, загружающей файл `.hdl`, который он должен тестировать. Симулятор всегда ищет этот файл в текущей папке.

В принципе, меню `File` симулятора позволяет пользователю интерактивно загружать как файл `.hdl`, так и файл сценария `.tst`, что допускает некоторые проблемы. Например, вы можете загрузить в симулятор файл `.hdl`, над которым работаете, а затем загрузить тестовый сценарий из другой папки. При запуске тестового скрипта он вполне может загрузить в симулятор другую версию HDL-программы (возможно, файл-заглушку). В случае сомнений посмотрите на панель с названием HDL в графическом интерфейсе симулятора и проверьте, какой именно HDL-код в него загружен.

Лучший практический совет: пользуйтесь меню `File` симулятора для загрузки либо файла `.hdl`, либо файла `.tst`, но не обоих сразу.

Тестирование микросхем в изоляции: в какой-то момент вы можете подумать, что ваша микросхема реализована правильно, несмотря на то что она все равно не проходит тест. И действительно, возможны случаи, когда в целом микросхема реализована идеально, но одна из ее частей при этом не работает. Кроме того, микросхема, ранее успешно прошедшая тест, может выйти из строя при использовании в качестве составной части другой микросхемы. Одно из самых больших ограничений инструментов по проектированию аппаратного обеспечения заключается в том, что сценарии тестирования — особенно те, которые тестируют сложные микросхемы — не могут гарантировать идеальной работы тестируемой микросхемы при любых обстоятельствах.

Положительный момент заключается в том, что вы всегда можете выяснить, какая часть микросхемы вызывает проблему. Создайте подпапку `test` и скопируйте в нее только три файла `.hdl`, `.tst` и `.out`, относящиеся к микросхеме, которую вы сейчас собираете. Если ваша реализация микросхемы проходит тест в этой подпапке в текущем виде (позволяя симулятору использовать встроенные микросхемы-части по умолчанию), то проблема в одной из ваших реализаций микросхем-частей, то есть в одной из микросхем, которые вы собрали ранее в этом проекте. Копируйте одну за другой другие микросхемы в эту тестовую папку и повторяйте тест, пока не выявите проблемную микросхему

Синтаксические ошибки HDL: симулятор аппаратуры отображает ошибки в нижней строке состояния. На компьютерах с маленькими экранами эти сообщения иногда уходят за нижнюю границу экрана и не видны. Если вы загрузили HDL-программу и в панели HDL ничего не отображается, а сообщения об ошибке не видно, то проблема, вероятно, именно в отображении сообщения об ошибке. На вашем компьютере должен быть предусмотрен способ перемещения окна с помощью клавиатуры. Например, в Windows нажмите одновременно клавиши `Alt + Space`, затем `M` и переместите окно клавишами со стрелками.

Несоединенные контакты: симулятор аппаратуры не считает контакты без соединений ошибками. По умолчанию для любого неподключенного входного или выходного контакта устанавливается значение `false` (двоичное значение 0). Это может привести к загадочным ошибкам в ваших реализациях микросхем.

Если значение на выходе вашей микросхемы всегда равно 0, убедитесь, что в вашей программе контакт выхода правильно подключен к другому контакту. В частности, дважды проверьте названия внутренних контактов («проводов»), которые прямо или косвенно соединены с этим выходом. Особенно здесь опасны опечатки, потому что при этом симулятор не отображает сообщения о том, что провода не соединены. Рассмотрим для примера высказывание `F00 (... , sum = sun)`, согласно которому выходной контакт `sum` микросхемы `F00` подключается к внутреннему контакту. В соответствии с этим высказыванием симулятор, несколько не смутившись, создаст внутренний контакт с именем `sun`. Но если на выход реализуемой микросхемы или на вход другой микросхемы-части должно подаваться значение с выхода `sum`, то на выходе *всегда* будет 0, потому что он ни с чем не соединен.

Короче говоря, если значение на выходном контакте всегда равно 0 или если одна из частей микросхемы работает неправильно, проверьте написание имен всех контактов и убедитесь, что все входные контакты микросхемы-части к чему-то подключены.

Настройка тестирования: для каждого файла вида `chip.hdl`, где *chip* — это название микросхемы, которую вы должны реализовать, в папке проекта содержится также поставляемый тестовый сценарий под названием `chip.tst` и файл сравнения под названием `chip.cmp`. После того как ваша микросхема в ходе проверки начнет выдавать какие-то данные, в этой папке также появится выходной файл под названием `chip.out`. Если ваша микросхема не справится с тестовым сценарием, не забудьте просмотреть файл `.out`. Просмотрите перечисленные выходные значения и подумайте, в чем может быть ошибка. Если по какой-то причине вы не сможете открыть выходной файл в графическом интерфейсе симулятора, его всегда можно открыть с помощью текстового редактора.

При желании можно провести собственные тесты. Скопируйте прилагаемый тестовый сценарий в другой файл — допустим, в файл *MyTestChip.tst* — и измените команды в нем, чтобы получить больше информации о поведении вашего чипа.

Для начала измените имя выходного файла в строке `output-file` и удалите строку `compare-to`. Это приведет к тому, что тест всегда будет выполняться до конца (по умолчанию симуляция останавливается, когда строка вывода расходится с соответствующей строкой в файле сравнения). Строку `output-list` можно настроить для отображения значений на внутренних контактах вашей микросхемы.

В приложении 3 содержится документация *Языка описания тестов (TDL)*, на котором пишутся все эти команды.

Создание шин из внутренних контактов (индексирование): индексирование внутренних контактов не допускается. Единственные контакты, для которых допускается создание шин (индексирование), — это входы и выходы реализуемой микросхемы или входные и выходные контакты ее составных частей. Но существует способ обойти это ограничение и проиндексировать внутренние контакты. Вот пример, который не работает:

```
CHIP Foo {
  IN in[16];
  OUT out;
  PARTS:
  Not16 (in=in, out=notIn);
  Or8Way (in=notIn[4..11], out=out); // Ошибка: нельзя
                                     индексировать
                                     внутреннюю
                                     шину
}
```

Возможный вариант решения этой проблемы:

```
Not16 (in=in, out[4..11]=notIn);
Or8Way (in=notIn, out=out); // Работает!
```

Множественные выходы: иногда требуется разделить многобитное значение шины-вывода на две шины. Это можно сделать при помощи нескольких привязок `out=`.

Например:

```
CHIP Foo {
  IN in[16];
  OUT out[8];
  PARTS:
  Not16 (in=in, out[0..7]=low8, out[8..15]=high8);
  // Разделение значения out
  Bar8Bit (a=low8, b=high8, out=out);
}
```

Иногда может возникнуть потребность в том, чтобы вывести значение на выход и одновременно использовать его для дальнейших вычислений. Это можно сделать следующим образом:

```
CHIP Foo {
  IN a, b, c;
  OUT out1, out2;
  PARTS:
  Bar (a=a, b=b, out=x, out=out1);    // Выход Bar идет
                                       на выход out1
                                       Foo
  Baz (a=x, b=c, out=out2);           // Выход Bar также
                                       служит источником
                                       для входа Baz
}
```

«Автозаполнение» (как бы) для микросхем-частей: стандартные высказывания для всех микросхем, упомянутых в этой книге, вместе с их названиями и названиями всех их контактов в скобках указаны в приложении 4, которое также имеет веб-версию (по адресу: www.nand2tetris.org). Чтобы использовать микросхему-часть в реализации

своей микросхемы, скопируйте соответствующий ей текст из онлайн-документа и вставьте его в свою HDL-программу, а затем вставьте недостающие названия контактов. Такая практика экономит время и сводит к минимуму ошибки при наборе текста.

Приложение 3. Язык описания тестов

Ошибки — врата открытия.

— Джеймс Джойс (1882–1941)

Тестирование — критически важный элемент разработки систем, которому обычно уделяется недостаточное внимание при обучении информатике. В практическом курсе «От Nand до “Тетриса”» мы относимся к тестированию очень серьезно. Мы считаем, что, прежде чем приступить к разработке нового аппаратного или программного модуля *P*, необходимо сначала разработать модуль *T*, предназначенный для его тестирования. Более того, этот модуль *T* должен быть частью контракта на разработку *P*. Поэтому для каждой микросхемы или программной системы, описанной в этой книге, мы предоставляем официальные, написанные нами тестовые программы. Конечно, вы можете тестировать свою работу так, как считаете нужным, но подразумевается, что согласно контракту ваша реализация должна пройти *наши* тесты.

Для того чтобы упростить определение и выполнение многочисленных тестов, разбросанных по всем проектам книги, мы разработали единый язык описания тестов. Он работает практически одинаково для всех инструментов, поставляемых в рамках практического курса «От Nand до “Тетриса”»: *симулятора аппаратуры*, используемого для моделирования и тестирования микросхем, написанных на языке HDL; *эмулятора процессора*, используемого для моделирования и тестирования программ на машинном языке; а также *эмулятора*

виртуальной машины, используемого для моделирования и тестирования программ, написанных на языке ВМ, которые обычно представляют собой скомпилированные программы на языке Jack.

Каждый из этих симуляторов имеет графический интерфейс, позволяющий тестировать загруженную микросхему или программу в интерактивном или пакетном режиме с помощью тестового сценария. Тестовый сценарий (скрипт) — это последовательность команд, которые загружают аппаратный или программный модуль в соответствующий симулятор и подвергают его серии заранее запланированных схем тестирования. Кроме того, тестовые сценарии содержат команды для печати результатов тестирования и сравнения их с желаемыми результатами, определенными в прилагаемых файлах сравнения. В целом тестовый сценарий позволяет систематически, воспроизводимо и документированно тестировать основной код — бесценное требование в любом проекте по разработке аппаратного или программного обеспечения.

В рамках практического курса «От Nand до “Тетриса”» мы не ожидаем, что учащиеся будут писать свои сценарии тестирования. Все тестовые сценарии, необходимые для тестирования всех упомянутых в книге аппаратных и программных модулей, поставляются вместе с материалами проекта. Таким образом, цель данного приложения — не научить вас писать тестовые скрипты, а помочь понять синтаксис и логику поставляемых тестовых скриптов. Конечно, при этом вы можете изменять прилагаемые сценарии и создавать новые по своему усмотрению.

П3.1. Общие рекомендации

Следующие рекомендации по использованию языка описания тестов применимы ко всем программным инструментам и сценариям тестирования.

Формат и использование файлов: в процессе тестирования аппаратного или программного модуля используются четыре типа файлов.

Хотя это и не обязательно, мы рекомендуем, чтобы все четыре файла имели одинаковый префикс (имя файла).

Xxx.yyy: где *Xxx* — имя тестируемого модуля, а *yyy* — *hdl*, *hack*, *asm* или *vm*, означающие, соответственно, микросхему, определенную с помощью языка HDL, программу, написанную на машинном языке Hack, программу, написанную на языке ассемблера Hack, или программу, написанную на языке VM.

Xxx.tst: тестовый сценарий, который проводит симулятор через ряд шагов, предназначенных для тестирования кода, хранящегося в *Xxx*.

Xxx.out: необязательный выходной файл, в который команды сценария могут записывать текущие значения выбранных переменных во время симуляции.

Xxx.spr: необязательный файл сравнения, содержащий *желаемые* значения выбранных переменных, то есть значения, которые *должна* генерировать симуляция, если модуль реализован правильно.

Все эти файлы должны храниться в одной папке, которую удобно назвать *Xxx*. В документации и описаниях всех симуляторов термин «текущая папка» означает папку, из которой был открыт последний файл в среде симулятора.

Пробелы и пустое пространство: пробелы, символы новой строки и комментарии в тестовых сценариях (файлы *Xxx.tst*) игнорируются. В тестовых сценариях могут присутствовать следующие форматы комментариев:

```
// Комментарий до конца строки
/* Комментарий до закрытия */
/** Комментарий для документации API */
```

Тестовые сценарии не чувствительны к регистру за исключением имен файлов и папок.

Использование: для каждого аппаратного или программного модуля *Xxx* в курсе «От Nand до “Тетриса”» мы предоставляем файл сценария *Xxx.tst* и файл сравнения *Xxx.cmp*. Эти файлы предназначены для тестирования вашей реализации *Xxx*. В некоторых случаях мы также поставляем каркасную версию *Xxx*, например, HDL-интерфейс с отсутствующей реализацией. Все файлы во всех проектах являются обычными текстовыми файлами, которые следует просматривать и редактировать с помощью обычных текстовых редакторов.

Как правило, сеанс моделирования начинается с загрузки прилагаемого файла сценария *Xxx.tst* в соответствующий симулятор. Первая команда в сценарии обычно загружает код, хранящийся в тестируемом модуле *Xxx*. Далее могут следовать команды, инициализирующие выходной файл и задающие файл сравнения. Оставшиеся команды сценария запускают собственно тесты.

Управление симуляцией: каждый из прилагаемых симуляторов имеет набор меню и значков (иконок) для управления симуляцией.

Меню «Файл» (File): позволяет загрузить в симулятор либо программу (файл *.hdl*, файл *.hack*, файл *.asm*, файл *.vm* или имя папки), либо тестовый сценарий (файл *.tst*). Если пользователь не загружает тестовый сценарий, симулятор загружает тестовый сценарий по умолчанию (описано ниже).

Значок «Проиграть»: приказывает симулятору выполнить следующий шаг моделирования, указанный в текущем загруженном сценарии тестирования.

Значок «Пауза»: приказывает симулятору приостановить выполнение текущего загруженного тестового сценария. Полезно для просмотра различных элементов моделируемой среды.

Значок «Перемотка вперед»: приказывает симулятору выполнить все команды в загруженном тестовом сценарии.

Значок «Стоп»: приказывает симулятору остановить выполнение загруженного тестового сценария.

Значок «Перемотка назад»: приказывает симулятору перезагрузить выполнение загруженного тестового сценария, то есть подготовиться к выполнению тестового сценария с первой команды и далее.

Обратите внимание, что перечисленные выше значки симулятора не «запускают код». Они скорее запускают тестовый сценарий, который выполняет код.

П3.2. Тестирование микросхем в симуляторе аппаратуры

Поставляемый в комплекте аппаратный симулятор предназначен для тестирования и моделирования описаний микросхем, реализованных на языке описания аппаратуры (HDL), описанном в приложении 2. Глава 1 содержит важную справочную информацию о разработке и тестировании микросхем; поэтому мы рекомендуем сначала прочитать ее.

Пример: на иллюстрации П2.1 в приложении 2 описана микросхема `Eq3`, предназначенная для проверки равенства трех 1-битных входов. На иллюстрации П3.1 представлены сценарий `Eq3.tst`, разработанный для тестирования микросхемы, и файл сравнения `Eq3.smp`, содержащий ожидаемый вывод этого теста.

Сценарий тестирования обычно начинается с некоторых команд настройки, за которыми следует серия шагов симуляции, каждый из которых заканчивается точкой с запятой. Шаг симуляции обычно приказывает симулятору связать входные контакты микросхемы с тестовыми значениями, проверить логику микросхемы и записать выбранные значения переменных в указанный выходной файл.

Микросхема `Eq3` имеет три 1-битных входа; таким образом, для полного теста потребуется восемь сценариев тестирования. Размер полного теста растет экспоненциально с увеличением размера входных данных. Поэтому большинство сценариев тестирования проверяют только подмножество репрезентативных входных значений, как показано на иллюстрации.

```
/* Eq3.tst: Тестирует программу. На выход чипа Eq3 подается 1, если на трех
его входах одинаковое значение, иначе 0. */

load Eq3.hdl,           // Загружает в симулятор программу HDL.
output-file Eq3.out,     // Записывает выходы скрипта в этот файл.
compare-to Eq3.cmp,      // Сравнивает выходы скрипта с этим файлом.
output-list a b c out;   // Каждая последующая команда output
                        // записывает значения переменных a, b, c
                        // и out в файл выхода.

set a 0, set b 0, set c 0, eval, output;
set a 1, set b 1, set c 1, eval, output;
set a 1, set b 0, set c 0, eval, output;
set a 0, set b 1, set c 0, eval, output;
set a 1, set b 0, set c 1, eval, output;
```

Eq3.cmp

a	b	c	out
0	0	0	1
1	1	1	1
1	0	0	0
0	1	0	0
1	0	1	0

Иллюстрация ПЗ.1. Тестовый сценарий и файл сравнения (пример).

Типы данных и переменные: тестовые сценарии поддерживают два типа данных: целые числа и строки. Целочисленные константы могут быть выражены в десятичном (префикс %D) формате, который используется по умолчанию, двоичном (префикс %B) или шестнадцатеричном (префикс %X). Эти значения всегда переводятся в эквивалентные им двоичные значения с дополнительным кодом. Рассмотрим, например, следующие команды:

```
set a1%B1111111111111111
set a2%XFFFF
set a3%D-1
set a4-1
```

Для всех четырех переменных установлено одно и то же значение: 1111111111111111 в двоичном формате, то есть –1 в десятичном формате.

Строковые значения задаются с помощью префикса %S и должны быть заключены в двойные кавычки. Строки используются исключительно для печати и не могут быть присвоены в качестве значения переменным.

Двухфазный генератор тактовой частоты (используется только при тестировании последовательностных микросхем) выдает серию значений, обозначаемых 0, 0+, 1, 1+, 2, 2+, 3, 3+ и т. д. Прогрессией этих *тактовых циклов* (также называемых *единицами времени*) можно управлять с помощью двух команд скрипта, `tick` и `tock`. `tick` сдвигает значение генератора с `t` до `t+`, а `tock` — с `t+` до `t + 1`, начиная следующую единицу времени. Текущая единица времени хранится в системной переменной `time`, доступной только для чтения.

Команды сценария могут обращаться к трем типам переменных: контактам (*pins*), переменным встроенных микросхем (*chips*) и системной переменной `time`.

Контакты: входные, выходные и внутренние контакты моделируемой микросхемы (например, команда `set in 0` присваивает значение 0 контакту с именем `in`).

Переменные встроенных микросхем: открываются внешней реализацией микросхемы (см. иллюстрацию П3.2).

Системная переменная time: количество единиц времени, прошедших с момента начала симуляции (переменная только для чтения).

Команды сценария: сценарий — это последовательность команд. Каждая команда завершается запятой, точкой с запятой или восклицательным знаком. Эти терминаторы имеют следующую семантику.

Запятая (,): завершает команду сценария.

Точка с запятой (;): завершает команду сценария и шаг симуляции.

Шаг симуляции состоит из одной или нескольких команд сценария. Когда пользователь с помощью меню симулятора включает пошаговый режим, симулятор выполняет сценарий с текущей команды до точки с запятой, после чего симуляция приостанавливается.

Восклицательный знак (!): завершает команду сценария и останавливает выполнение сценария. В дальнейшем пользователь может возобновить выполнение сценария с этого момента. Обычно используется для отладки.

Ниже мы сгруппируем команды сценария в два концептуальных раздела: *установочные команды*, используемые для загрузки файлов и инициализации настроек, и *команды симуляции*, используемые для выполнения собственно тестов.

Установочные команды

load *Xxx.hdl*: загружает в симулятор программу HDL, хранящуюся в файле *Xxx.hdl*. Имя файла должно включать расширение *.hdl* и не должно включать спецификацию пути. Симулятор попытается загрузить файл из текущей папки и, если это не удастся, из папки *tools/builtInChips*.

output-file *Xxx.out*: указывает симулятору записывать результаты команд вывода в названный файл, который должен содержать расширение *.out*. Выходной файл будет создан в текущей папке.

output-list *v1, v2, ...*: указывает, что записывать в выходной файл, когда в сценарии встречается команда вывода (до следующей команды *output-list*, если она есть). Каждое значение в списке — имя переменной согласно формальным спецификациям. Команда также выдает одну строку заголовка, состоящую из имен переменных, которая записывается в выходной файл. Каждый элемент *v* в выходном списке имеет синтаксис *varName format padL.len.padR* (без пробелов). Эта директива предписывает симулятору записать *padL* пробелов, затем текущее значение переменной *varName* в указанном формате и длиной в *len* столбцов, затем *padR* пробелов и, наконец, символ разделителя *|*. *Формат* может быть либо *%B* (двоичный), *%X* (шестнадцатеричный), *%D* (десятичный), либо *%S* (строковый). По умолчанию используется формат *%B1.1.1*.

Например, микросхема *CPU.hdl* платформы *Nack* имеет вход с именем *reset*, выход с именем *pc* (помимо прочих) и микросхему-часть с именем *DRegister* (помимо прочих). Если мы

хотим отслеживать значения этих элементов во время моделирования, то можно воспользоваться примерно следующей командой:

```
Output-list  time%S1.5.1          // Системная переменная
                                     time
               reset%B2.1.2        // Один из входных
                                     контактов чипа
               pc%D2.3.1           // Один из выходов чипа
               DRegister[]%X3.4.4 // Внутреннее состояние
                                     этого чипа
```

Переменные состояния встроенных микросхем объясняются ниже. Эта команда `output-list` может выдать следующий результат после двух последующих команд вывода `output`:

	time		reset		pc		DRegister[]	
	20+		0		21		FFFF	
	21		0		22		FFFF	

compare-to Xxx.cmp: указывает, что строка вывода, генерируемая каждой последующей командой вывода, должна сравниваться с соответствующей строкой в указанном файле сравнения (который должен содержать расширение `.cmp`). Если какие-либо две строки не совпадают, симулятор выводит сообщение об ошибке и останавливает выполнение сценария. Предполагается, что файл сравнения находится в текущей папке.

Команды симуляции

set varName value: присваивает значение *value* переменной *varName*. Переменная связана либо с каким-либо контактом микросхемы, либо с контактами части микросхемы.

eval: дает команду симулятору применить логику микросхемы к текущим значениям входных выводов и вычислить выходные значения.

output: заставляет симулятор выполнить следующие действия.

1. Получить текущие значения всех переменных, перечисленных в последней команде `output-list`.
2. Создать выходную строку в формате, указанном в последней команде `output-list`.
3. Записать выходную строку в выходной файл.
4. (Если файл сравнения был предварительно объявлен с помощью команды `compare-to`): если выходная строка отличается от текущей строки файла сравнения, вывести сообщение об ошибке и остановить выполнение скрипта.
5. Переместить на новые строки курсоры выходного файла и файла сравнения.

tick: завершает первую фазу текущей единицы времени (тактового цикла).

tock: завершает вторую фазу текущей единицы времени и начинает первую фазу следующей единицы времени.

repeat *n* {команды}: приказывает симулятору повторить команды, заключенные в фигурные скобки, *n* раз. Если *n* опущено, симулятор повторяет команды до тех пор, пока симуляция не будет остановлена по какой-либо причине (например, когда пользователь нажимает на значок «Стоп»).

while *booleanCondition* {команды}: приказывает симулятору повторять команды, заключенные в фигурные скобки, до тех пор, пока условие *booleanCondition* сохраняется истинным. Условие имеет вид *x or y*, где *x* и *y* — константы или имена переменных, а *or* это `=`, `>`, `<`, `>=`, `<=` или `<>`. Если *x* и *y* — строки, *or* может быть либо `=`, либо `<>`.

echo *text*: отображает текст в строке состояния симулятора. Текст должен быть заключен в двойные кавычки.

clear-echo: очищает строку состояния симулятора.

breakpoint varName value: начинает сравнивать текущее значение указанной переменной *varName* с указанным значением *value* после выполнения каждой последующей команды сценария. Если переменная содержит указанное значение *value*, выполнение останавливается и выводится сообщение. В противном случае выполнение продолжается как обычно. Полезно для целей отладки.

clear-breakpoints: очищает все ранее определенные точки breakpoint.

builtInChipName method argument(s): выполняет указанный метод *method* указанной встроенной микросхемы-части с использованием указанных аргументов *argument(s)*. Разработчик встроенной микросхемы может предоставить методы, которые позволят пользователю (или тестовому сценарию) манипулировать симулируемой микросхемой. См. иллюстрацию ПЗ.2.

Переменные встроенных микросхем: микросхемы реализуются либо HDL-программами, либо внешне поставляемыми исполняемыми модулями. В последнем случае микросхема считается *встроенной*. Доступ к состоянию встроенных микросхем можно получить с помощью синтаксиса *chipName[varName]*, где *varName* — зависящая от реализации переменная, которая должна быть указана в API микросхемы. См. иллюстрацию ПЗ.2.

Рассмотрим, например, команду сценария `set RAM16K[1017] 15`. Если RAM16K — это текущая моделируемая микросхема или часть моделируемой микросхемы, то эта команда присваивает ячейке памяти 1017 значение 15. А поскольку встроенный чип RAM16K имеет побочные эффекты графического интерфейса, новое значение также отразится в визуальном изображении чипа.

Если встроенный чип поддерживает внутреннее состояние с одним значением, доступ к текущему состоянию можно получить через запись *chipName[]*. Если внутреннее состояние — вектор, используется запись *chipName[i]*. Например, при моделировании встроенной микросхемы Register можно записать команды сценария типа `set`

Register[] 135. Эта команда записывает в микросхему значение 135; в следующую единицу времени микросхема Register фиксирует это значение, которое начинает выдавать ее контакт на выходе.

Имя чипа	Соответствующие переменные	Тип/диапазон данных	Методы
Register	Register[]	16-битное (−32768...32767)	
ARegister	ARegister[]	16-битное	
DRegister	DRegister[]	16-битное	
PC (про- граммный счетчик)	PC[0..]	15-битное	
RAM8	RAM8[0..7]	каждое значение 16-битное	
RAM64	RAM64[0..63]	каждое значение 16-битное	
RAM512	RAM512[0..511]	каждое значение 16-битное	
RAM4K	RAM4K[0..4095]	каждое значение 16-битное	
RAM16K	RAM16K[0..16383]	каждое значение 16-битное	
ROM32	ROM32[0..32767]	каждое значение 16-битное	load Xxx.hack / Xxx.asm
Screen	Screen[0..16383]	каждое значение 16-битное	
Keyboard	Keyboard[]	16-битное, только для чтения	

Иллюстрация ПЗ.2. Переменные и методы ключевых встроенных чипов «От Nand до “Тетриса”».

Методы встроенных микросхем: встроенные микросхемы также могут иметь *методы*, которые можно отображать в командах сценариев. Например, в компьютере Nask программы размещаются в блоке памяти команд, реализованном встроенной микросхемой ROM32K. Перед запуском программы на машинном языке на компьютере Nask

программа должна быть загружена в эту микросхему. Для облегчения задачи встроенная реализация ROM32K имеет метод загрузки, позволяющий загрузить текстовый файл с командами машинного языка. Доступ к этому методу можно получить с помощью скриптовой команды ROM32K `load fileName.hack`.

Завершающий пример: приведем под конец относительно сложный тестовый сценарий, предназначенный для тестирования самой верхней в иерархии микросхем компьютера Hack микросхемы Computer.

Один из способов тестирования микросхемы Computer — это загрузка в нее программы на машинном языке и наблюдение за выбранными значениями по мере выполнения компьютером программы, по одной инструкции за раз. Допустим, мы написали программу на машинном языке, которая вычисляет максимальное значение RAM[0] и RAM[1] и записывает результат в RAM[2]. Программа хранится в файле с именем Max.hack.

Обратите внимание, что на том низком уровне, на котором мы работаем, если такая программа не запускается как нужно, то ошибки могут быть как в программе, так и в аппаратном обеспечении (или, возможно, в тестовом скрипте или в симуляторе аппаратуры). Для простоты предположим, что ошибок нет, разве что в самом смоделированном чипе Computer.

Для тестирования микросхемы Computer с помощью программы Max.hack мы написали тестовый скрипт под названием ComputerMax.tst. Этот скрипт загружает в симулятор аппаратуры файл Computer.hdl, а затем загружает программу Max.hack в ROM32K, представляющую собой часть тестируемой микросхемы. Разумный способ проверить, правильно ли работает микросхема, заключается в следующем: поместить некоторые значения в RAM[0] и RAM[1], перезагрузить компьютер, запустить генератор тактовой частоты на достаточное количество циклов и проверить RAM[2]. Коротче говоря, именно для этого и предназначен сценарий, показанный на иллюстрации ПЗ.3.

Как определить, хватит ли четырнадцати тактов для выполнения данной программы? Это можно выяснить методом проб и ошибок,

начав с относительно большого значения и наблюдая за тем, как со временем стабилизируется значение на выходах компьютера, либо анализируя поведение загруженной программы во время выполнения.

```
/* Тестовый сценарий ComputerMax.tst.
Использует программу Max.hack, записывающую
в RAM[2] максимальное из значений (RAM[0], RAM[1]). */
// Загружает Computer и подготавливает симуляцию:
load Computer.hdl,
output-file ComputerMax.out,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];
// Загружает Max.hack в микросхему-часть ROM32K:
ROM32K load Max.hack,
// Записывает в первые 2 ячейки части RAM16K
// тестовые значения:
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
// Совершает достаточное количество циклов
// для завершения выполнения программы:
repeat 14 {
    tick, tock,
    output;
}
// (Сценарий продолжается справа.)
```

```
// Подготавливает другой тест с другими
// значениями. Перезагружает Computer:
// устанавливает reset на 1, запускает
// генератор такт. частоты, чтобы перенастроить
// программный счетчик
// (PC, последовательностный чип) на новое
// значение перезагрузки:
set reset 1,
tick,
tock,
output;
// Устанавливает reset на 0, загружает новые
// тестовые значения и совершает достаточное
// количество циклов для завершения
// выполнения программы:
set reset 0,
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
repeat 14 {
    tick, tock,
    output;
}
```

Иллюстрация ПЗ.3. Тестирование самой верхней в иерархии микросхемы Computer.

Тестовые сценарии по умолчанию: в каждом симуляторе «От Nand до «Тетриса»» имеются тестовые сценарии по умолчанию. Если пользователь не загружает в симулятор тестовый сценарий, использует сценарий по умолчанию. Для симулятора аппаратуры сценарий по умолчанию следующий:

```
//Тестовый сценарий симулятора аппаратуры
по умолчанию:
repeat {
    tick,
    tock;
}
```

П3.3. Тестирование программ на машинном языке в симуляторе ЦПУ

В отличие от *симулятора аппаратуры*, программы общего назначения, предназначенной для поддержки построения любой аппаратной платформы, прилагаемый *эмулятор ЦПУ* (*эмулятор процессора*) представляет собой одноцелевой инструмент, предназначенный для симуляции выполнения программ на машинном языке конкретной платформы: компьютера Hack. Программы составляются либо на символьном, либо на двоичном машинном языке Hack, описанном в главе 4.

Как обычно, симуляция включает четыре файла: тестируемую программу (*Xxx.asm* или *Xxx.hack*), тестовый сценарий (*Xxx.tst*), дополнительный выходной файл (*Xxx.out*) и дополнительный файл сравнения (*Xxx.cmp*). Все эти файлы располагаются в одной папке обычно с именем *Xxx*.

Пример: рассмотрим программу умножения *Mult.hack*, предназначенную для выполнения $RAM[2] = RAM[0] * RAM[1]$. Предположим, мы хотим протестировать эту программу в эмуляторе процессора. Разумный способ тестирования заключается в следующем: поместим некоторые значения в *RAM[0]* и *RAM[1]*, запустим программу и проверим *RAM[2]*. Логiku этих действий выполняет тестовый скрипт, показанный на иллюстрации П3.4.

Переменные. Команды сценария, запущенные на эмуляторе процессора, могут получать доступ к следующим элементам компьютера Hack.

- A: Текущее значение адресного регистра (беззнаковое 15-битное).
- D: Текущее значение регистра данных (16-битное).
- PC: Текущее значение программного счетчика (беззнаковое 15-битное).
- RAM[i]: Текущее значение RAM по адресу *i* (16-битное).
- time: Количество *единиц времени* (также называемых *тактовыми циклами*), прошедших с начала симуляции (переменная только для чтения).

Команды: эмулятор ЦПУ поддерживает все команды, описанные в разделе П3.2, за исключением следующих изменений.

load progName: где *progName* — это либо *Xxx.asm*, либо *Xxx.hack*.

Эта команда загружает в симулируемую память команд программу (подлежащую тестированию) на машинном языке. Если программа написана на языке ассемблера, симулятор переводит ее в двоичный формат на ходу, в процессе выполнения команды *load progName*.

eval: не применяется в эмуляторе ЦПУ.

builtInChipName аргумент(ы) метода: не применяется в эмуляторе ЦПУ.

tickTock: эта команда используется вместо команд *tick* и *tock*.

Каждая команда переводит генератор тактовой частоты на одну единицу времени (тактыый цикл).

```
// Загружает программу и подготавливает симуляцию:
load Mult.hack,
output-file Mult.out,
compare-to Mult.cmp,
output-list RAM[2]%D2.6.2;

// Устанавливает тестовые значения первых 2 ячеек RAM:
set RAM[0] 2,
set RAM[1] 5;

// Совершает достаточное количество циклов для выполнения программы:
repeat 20 {
    ticktock;
}
output;

// Перезапускает программу с другими тестовыми значениями:
set PC 0,
set RAM[0] 8,
set RAM[1] 7;

// Mult.hack основана на примитивном алгоритме умножения,
// поэтому умножение больших чисел требует больше циклов:
repeat 50 {
    ticktock;
}
output;
```

Иллюстрация П3.4. Тестирование программы на машинном языке в эмуляторе ЦПУ.

Тестовый сценарий по умолчанию

```
//Тестовый сценарий симулятора ЦПУ по умолчанию:  
repeat {  
    ticktock;  
}
```

П3.4. Тестирование программ ВМ в эмуляторе ВМ

Прилагаемый *эмулятор виртуальной машины (эмулятор ВМ)* представляет собой Java-реализацию виртуальной машины, описанной в главах 7–8. Его можно использовать для симуляции выполнения программ ВМ, визуализации их операций и отображения состояния задействованных сегментов виртуальной памяти.

Программа ВМ состоит из одного или нескольких файлов `.vm`. Таким образом, имитация ВМ-программы включает тестируемую программу (один файл `Xxx.vm` или папку `Xxx`, содержащую один или несколько файлов `.vm`) и опционально тестовый сценарий (`Xxx.tst`), файл сравнения (`Xxx.cmp`) и выходной файл (`Xxx.out`). Все эти файлы располагаются в одной папке обычно с именем `Xxx`.

Сегменты виртуальной памяти: команды ВМ `push` и `pop` предназначены для работы с *сегментами виртуальной памяти* (`argument`, `local` и т. д.). Эти сегменты должны быть выделены в оперативной памяти хоста, и эту задачу выполняет эмулятор ВМ в качестве побочного эффекта симуляции выполнения команд ВМ `call`, `function` и `return`.

Код запуска: транслируя программу ВМ, транслятор ВМ генерирует код машинного языка, устанавливающий указатель стека на 256 и вызывающий функцию `Sys.init`, которая затем инициализирует классы ОС и вызывает `Main.main`. Аналогичным образом, когда эмулятору ВМ дается указание выполнить программу ВМ (набор одной или нескольких функций ВМ), он программируется на запуск функции

`sys.init`. Если такая функция в загруженном коде ВМ не находится, эмулятор программируется на начало выполнения первой команды в загруженном коде ВМ.

Последняя особенность была добавлена в эмулятор ВМ для поддержки модульного тестирования транслятора ВМ, которому посвящены две главы книги и два проекта. В проекте 7 мы создаем базовый транслятор виртуальной памяти, обрабатывающий только команды `push`, `pop` и арифметические команды без обработки команд вызова функций. Для выполнения таких программ нужно каким-то образом привязать сегменты виртуальной памяти к ОЗУ (RAM) хоста — по крайней мере, те сегменты, которые упоминаются в имитируемом коде ВМ. Такую инициализацию удобно выполнить с помощью команд сценария, управляющих базовыми адресами виртуальных сегментов оперативной памяти. Используя эти команды, мы можем привязать виртуальные сегменты к любому участку ОЗУ хоста.

Пример: файл `FibonacciSeries.vm` содержит последовательность команд ВМ, вычисляющих первые n элементов ряда Фибоначчи. Код рассчитан на работу с двумя аргументами: n и начальный адрес памяти, в которой должны храниться вычисленные элементы. На иллюстрации П3.5 показан тестовый сценарий, тестирующий эту программу при аргументах 6 и 4000.

Переменные: команды сценариев, запущенные на эмуляторе ВМ, могут получать доступ к следующим элементам ВМ.

Содержимое сегментов ВМ:

<code>local[i]:</code>	значение i -го элемента сегмента <code>local</code>
<code>argument[i]:</code>	значение i -го элемента сегмента <code>argument</code>
<code>this[i]:</code>	значение i -го элемента сегмента <code>this</code>
<code>that[i]:</code>	значение i -го элемента сегмента <code>that</code>
<code>temp[i]:</code>	значение i -го элемента сегмента <code>temp</code>

Указатели сегментов VM:

local: базовый адрес сегмента local RAM
 argument: базовый адрес сегмента argument RAM
 this: базовый адрес сегмента this RAM
 that: базовый адрес сегмента that RAM

```
/* Программа FibonacciSeries.vm program вычисляет первые n элементов ряда Фибоначчи.
В этом тесте n = 6, и значения записываются в ячейки RAM с адресами с 4000 по 4005. */

load FibonacciSeries.vm,
output-file FibonacciSeries.out,
compare-to FibonacciSeries.cmp,
output-list RAM[4000]%D1.6.2 RAM[4001]%D1.6.2 RAM[4002]%D1.6.2
          RAM[4003]%D1.6.2 RAM[4004]%D1.6.2 RAM[4005]%D1.6.2;

// Код программы не содержит команд function/call/return.
//Поэтому сценарий инициализирует стек и сегменты local и argument явно:
set SP 256,
set local 300,
set argument 400;

// Задаёт первый аргумент n = 6, второй аргумент для адреса, с которого начнут
// записываться значения, и совершает достаточно шагов VM для выполнения программы:

set argument[0] 6,
set argument[1] 4000;
repeat 140 {
  vmstep;
}
output;
```

Иллюстрация ПЗ.5. Тестирование программы VM в эмуляторе VM.

Специфичные для реализации переменные:

RAM[i]: значение i-го участка RAM хоста
 SP: значение указателя стека
 currentFunction: имя выполняемой в настоящий момент функции (только для чтения)
 line: содержит строку вида *currentFunctionName.lineIndexInFunction* (только для чтения). Например, когда выполнение доходит до третьей строки функции *Sys.init*, переменная *line* содержит значение *Sys.init.3*.

Можно использовать для установки контрольных точек в выбранных местах загруженной программы ВМ.

Команды: эмулятор ВМ поддерживает все команды, описанные в разделе П3.2, за исключением следующих изменений.

`load source`: где опциональный параметр *source* — либо *Xxx.vm*, то есть файл с кодом ВМ, либо *Xxx*, то есть имя папки, содержащей один или несколько файлов *.vm* (которые загружаются один за другим). Если файлы *.vm* расположены в текущей папке, аргумент *source* может опустаться.

`tick / tock`: не применяется.

`vmstep`: симулирует выполнение одной команды ВМ и переходит к следующей команде в коде.

Сценарий по умолчанию

//Сценарий эмулятора ВМ по умолчанию:

```
repeat {  
    vmStep;  
}
```

Приложение 4. Набор микросхем Nask

Микросхемы отсортированы в алфавитном порядке. Онлайн-версия этого документа с API-форматом, доступная по адресу: www.nand2tetris.org, пригодится во время работы над разработкой микросхем: для этого достаточно скопировать сигнатуры микросхемы в программу HDL и заполнить недостающие привязки (также называемые соединениями).

```
Add16(a=, b=, out=) /* Складывает два 16-битных значения
                        с дополнительным кодом */
ALU(x=, y=, zx=, nx=, zy=, ny=, f=, no=, out=, zr=, ng=) /* Hack ALU */
And(a=, b=, out=) /* вентиль And */
And16(a=, b=, out=) /* 16-битный And */
ARegister(in=, load=, out=) /* Адресный регистр (встроенная) */
Bit(in=, load=, out=) /* 1-битный регистр */
CPU(inM=, instruction=, reset=, outM=, writeM=, addressM=, pc=) /* Hack
                                                                CPU */
DFF(in=, out=) /* вентиль триггер (встроенная) */
DMux(in=, sel=, a=, b=) /* Направляет вход на один из двух выходов */
DMux4Way(in=, sel=, a=, b=, c=, d=) /* Направляет вход на один
                                   из четырех выходов */
DMux8Way(in=, sel=, a=, b=, c=, d=, e=, f=, g=, h=) /* Направляет вход
                                                    на один из восьми
                                                    выходов */
DRegister(in=, load=, out=) /* Регистр данных (встроенная) */
HalfAdder(a=, b=, sum=, carry=) /* Складывает два бита */
FullAdder(a=, b=, c=, sum=, carry=) /* Складывает три бита */
Incl6(in=, out=) /* Устанавливает на выходе out значение in + 1 */
```

```

Keyboard(out=) /* Карта памяти клавиатуры (встроенная) */
Memory(in=, load=, address=, out=) /* Память данных платформы Nask
                                   (RAM) */
Mux(a=, b=, sel=, out=) /* Выбирает один из двух входов */
Mux16(a=, b=, sel=, out=) /* Выбирает один из двух 16-битных входов */
Mux4Way16(a=, b=, c=, d=, sel=, out=) /* Выбирает один из четырех
                                       16-битных входов */
Mux8Way16(a=, b=, c=, d=, e=, f=, g=, h=, sel=, out=) /* Выбирает один
                                                         из восьми
                                                         16-битных входов
                                                         inputs */

Nand(a=, b=, out=) /* вентиль Nand (встроенная) */
Not16(in=, out=) /* 16-битный Not */
Not(in=, out=) /* вентиль Not */
Or(a=, b=, out=) /* вентиль Or */
Or8Way(in=, out=) /* 8-входовый Or */
Or16(a=, b=, out=) /* 16-битный Or */
PC(in=, load=, inc=, reset=, out=) /* Программный счетчик */
RAM8(in=, load=, address=, out=) /* 8-словная RAM */
RAM64(in=, load=, address=, out=) /* 64-словная RAM */
RAM512(in=, load=, address=, out=) /* 512-словная RAM */
RAM4K(in=, load=, address=, out=) /* 4K RAM */
RAM16K(in=, load=, address=, out=) /* 16K RAM */
Register(in=, load=, out=) /* 16-битный регистр */
ROM32K(address=, out=) /* Память команд платформы Nask (ROM,
встроенная) */
Screen(in=, load=, address=, out=) /* Карта памяти экрана (встроенная) */
Xor(a=, b=, out=) /* вентиль Xor */

```

Приложение 5. Набор символов Nask

32: пробел	56: 8	80: P	104: h	127: DEL
33: !	57: 9	81: Q	105: i	128: newLine
34: "	58: :	82: R	106: j	129: backSpace
35: #	59: ;	83: S	107: k	130: leftArrow
36: \$	60: <	84: T	108: l	131: upArrow
37: %	61: \	85: U	109: m	132: rightArrow
38: &	62: >	86: V	110: n	133: downArrow
39: '	63: ?	87: W	111: o	134: home
40: (	64: @	88: X	112: p	135: end
41:)	65: A	89: Y	113: q	136: pageUp
42: *	66: B	90: Z	114: r	137: pageDown
43: +	67: C	91: [	115: s	138: insert
44: ,	68: D	92: /	116: t	139: delete
45: -	69: E	93:]	117: u	140: esc
46: .	70: F	94: ^	118: v	141: f1
47: /	71: G	95: _	119: w	142: f2
48: 0	72: H	96: `	120: x	143: f3
49: 1	73: I	97: a	121: y	144: f4
50: 2	74: J	98: b	122: z	145: f5
51: 3	75: K	99: c	123: {	146: f6
52: 4	76: L	100: d	124:	147: f7
53: 5	77: M	101: e	125: }	148: f8
54: 6	78: N	102: f	126: ~	149: f9
55: 7	79: O	103: g		150: f10
				151: f11
				152: f12

Приложение 6. API ОС Jack

Язык Jack поддерживается восемью стандартными классами, которые предоставляют основные сервисы ОС, такие как распределение памяти, математические функции, ввод и вывод данных. В этом приложении задокументированы API этих классов.

Math

Этот класс предоставляет необходимые и часто встречающиеся математические функции.

`function int multiply(int x, int y):` возвращает произведение x и y . Обнаруживая в коде программы оператор умножения `*`, компилятор Jack обрабатывает его, вызывая эту функцию. Таким образом, выражения $x*y$ и вызов функции `Math.multiply(x, y)` в языке Jack возвращают одно и то же значение.

`function int divide(int x, int y):` возвращает целую часть x/y . Обнаруживая в коде программы оператор деления `/`, компилятор Jack обрабатывает его, вызывая эту функцию. Таким образом, Jack-выражения x/y и вызов функции `Math.divide(x, y)` возвращают одно и то же значение.

`function int min(int x, int y):` возвращает минимальное из значений x и y .

`function int max(int x, int y):` возвращает максимальное из значений x и y .

`function int sqrt(int x):` возвращает целочисленную часть квадратного корня из `x`.

String

Этот класс представляет строки, состоящие из значений символьного типа `char`, и предоставляет столь часто необходимые сервисы по обработке строк.

`constructor String new(int maxLength):` создает новую пустую строку с максимальной длиной `maxLength` и начальной длиной 0.

`method void dispose():` утилизирует (удаляет из памяти) эту строку.

`method int length():` возвращает количество символов в этой строке.

`method char charAt(int i):` возвращает *i*-й символ этой строки.

`method void setCharAt(int i, char c):` присваивает значение `c` *i*-му символу этой строки.

`method String appendChar(char c):` дополняет строку символом `c` и возвращает ее.

`method void eraseLastChar():` удаляет последний символ из этой строки.

`method int intValue():` возвращает целочисленное значение этой строки, пока не будет обнаружен нецифровой символ.

`method void setInt(int val):` задает этой строке указанное значение.

`function char backSpace():` возвращает символ `backspace`.

`function char doubleQuote():` возвращает символ двойных кавычек.

`function char newLine():` возвращает символ новой строки.

Array

В языке Jack массивы реализуются как экземпляры класса `OS Array`. После объявления к элементам массива можно обращаться с помощью синтаксиса `arr[i]`. Массивы в языке Jack нетипизированы: каждый элемент массива может содержать примитивный тип данных или тип объекта, причем разные элементы одного и того же массива могут иметь разные типы.

`function Array new(int size):` создает новый массив заданного размера.

`method void dispose():` утилизирует данный массив.

Output

Этот класс предоставляет функции для отображения символов. Предполагается, что экран, настроенный на отображение символов, состоит из 23 строк (с индексами 0...22, сверху вниз) по 64 символа в каждой (с индексами 0...63, слева направо). Верхнее левое расположение символов на экране имеет индекс (0, 0). Каждый символ отображается с помощью прямоугольного изображения 11 пикселей в высоту и 8 пикселей в ширину (включая поля для интервалов между символами и между строками). При необходимости растровые изображения («шрифт») всех символов можно найти, просмотрев заданный код класса `Output`. Видимый курсор, реализованный в виде небольшого заполненного квадрата, указывает, где будет отображаться следующий символ.

`function void moveCursor(int i, int j):` перемещает курсор на *j*-й столбец *i*-й строки с замещением отображаемого там символа.

`function void printChar(char c):` отображает символ в месте курсора и перемещает курсор на один столбец вперед.

`function void printString(String s):` отображает строковое значение, начиная с позиции курсора, и перемещает соответствующим образом курсор.

`function void printInt(int i):` отображает целое число, начиная с позиции курсора, и перемещает соответствующим образом курсор.

`function void println():` перемещает курсор в начало следующей строки.

`function void backSpace():` перемещает курсор на один столбец назад.

Screen

Этот класс предоставляет функции для отображения графических фигур на экране. Физический экран Hack состоит из 256 строк (с индексами 0...255, сверху вниз) по 512 пикселей в каждой (с индексами 0...511, слева направо). Левый верхний пиксель на экране имеет индекс (0, 0).

`function void clearScreen():` очищает весь экран.

`function void setColor(boolean b):` устанавливает текущий цвет. Этот цвет будет использоваться во всех последующих вызовах функции `drawXxx`. Черный цвет обозначается `true`, белый — `false`.

`function void drawPixel(int x, int y):` рисует пиксель (x, y) текущего цвета.

`function void drawLine(int x1, int y1, int x2, int y2):` рисует линию текущего цвета от пикселя (x1, y1) до пикселя (x2, y2).

`function void drawRectangle(int x1, int y1, int x2, int y2):` рисует заполненный текущим цветом прямоугольник с левым верхним углом в точке (x1, y1) и правым нижним в точке (x2, y2).

`function void drawCircle(int x, int y, int r):`рисует заполненную текущим цветом окружность радиуса $r \leq 181$ с центром в точке (x, y) .

Keyboard

Этот класс предоставляет функции для чтения вводимых данных со стандартной клавиатуры.

`function char keyPressed():`возвращает символ текущей нажатой клавиши на клавиатуре; если ни одна клавиша в данный момент не нажата, возвращает 0. Распознает все значения из набора символов Hack (см. приложение 5). К ним относятся символы `newline` (128, возвращаемое значение `String.newLine()`), `backSpace` (129, возвращаемое значение `String.backSpace()`), `leftArrow` (130), `upArrow` (131), `rightArrow` (132), `downArrow` (133), `home` (134), `end` (135), `pageUp` (136), `pageDown` (137), `insert` (138), `delete` (139), `esc` (140) и `f1-f12` (141–152).

`function char readChar():`ожидает нажатия и отпущения клавиши клавиатуры, затем выводит соответствующий символ на экран и возвращает его.

`function String readLine(String message):`отображает сообщение, считывает с клавиатуры введенную строку символов, пока не будет обнаружен символ `newline`, отображает строку и возвращает ее. Также обрабатывает вводимые пользователем обратные пробелы (`backspace`).

`function int readInt(String message):`выводит сообщение, считывает с клавиатуры введенную строку символов до обнаружения символа `newline`, выводит строку на экран и возвращает ее целочисленное значение до обнаружения первого нецифрового символа в введенной строке. Также обрабатывает вводимые пользователем обратные пробелы.

Memory

Этот класс предоставляет сервисы управления памятью. Оперативная память Hack RAM состоит из 32 768 слов, каждое из которых содержит 16-разрядное двоичное число.

`function int peek(int address):` возвращает значение `RAM[address]`.

`function void poke(int address, int value):` записывает указанное значение в `RAM[address]`.

`function Array alloc(int size):` находит доступный блок оперативной памяти заданного размера и возвращает его базовый адрес.

`function void deAlloc(Array o):` утилизирует заданный объект, который приводится как массив. Иными словами, делает блок оперативной памяти, начинающийся с этого адреса, доступным для распределения памяти в дальнейшем.

Sys

Этот класс предоставляет основные сервисы по выполнению программы.

`function void halt():` останавливает выполнение программы.

`function void error(int errorCode):` выводит код ошибки в формате `ERR<errorCode>` и останавливает выполнение программы.

`function void wait(int duration):` выжидает примерно миллисекунду и возвращается.

Алфавитный указатель

- AND-вентиль 36, 52
- Common Language Runtime (CLR) 237
- Complex Instruction Set Computing, CISC 171
- DFF, Data flip-flop. См. Триггер
- GPU. См. Графический процессор (GPU)
- HDL. См. Язык описания аппаратуры (HDL)
- Java Runtime Environment (JRE) 208, 236
- Last-in-first-out (LIFO). См. Последним пришел — первым вышел (LIFO)
- Nand-вентиль 25, 26, 429
- булева функция 50
- в реализации триггера 102
- как основа реализации аппаратного оборудования 60
- спецификация 51
- Nor-вентиль 64, 431
- Not-вентиль 36
- NOT-вентиль 51
- Ог-вентиль 36
- OR-вентиль 52
- Peek и poke 395, 420
- Python Virtual Machine (PVM) 237
- RAM (Random Access Memory). См. Оперативное запоминающее устройство (ОЗУ)
- RISC (Reduced Instruction Set Computing, архитектура с сокращенным набором команд) 171
- ROM (Read Only Memory). См. Постоянное запоминающее устройство (ПЗУ)
- XML 304, 325, 331
- Хог-вентиль 43, 47
- XOR-вентиль 52
- Ада. См. Кинг-Ноэль, Августа Ада
- Адресация 89, 146
- Адрес возврата 242, 251, 260, 261
- Адресное пространство 142, 158
- Адресный регистр 114, 117, 149, 152
- Алгоритм выделения памяти (базовый) 393
- АЛУ. См. Арифметико-логическое устройство (АЛУ)
- Арифметико-логические команды 239, 265
- Арифметико-логическое устройство (АЛУ) 27, 28, 77, 148, 161
- Архитектура фон Неймана 119, 143, 145

- Ассемблер 33, 173
API 183
двухпроходной ассемблер 181
макрокоманды 142, 191
мнемоники 174
общие принципы 174
псевдоинструкции 177
реализация 182
таблица символов 180
трансляция 180
- Булева алгебра 36, 425
- Булева арифметика 67
арифметические операции 68
булевы операторы 36
двоичное сложение 71
двоичные числа 68
двоичные числа со знаком 72
дополнительный код 72
метод дополнения до двух 72
отрицательные числа 72
переполнение 71
реализация 82
ускоренный перенос 85
фиксированный размер машинного слова 70
- бустрэппинг 195
- Ввод/вывод
привязка к памяти (отображение на память) 150
устройства ввода/вывода 131, 150
- Ввод с клавиатуры 401
обнаружение ввода 402
чтение одного символа 403
чтение строки 403
- Вентили. См. Логические вентили
- Ветвление 113, 121, 243
в машинном языке 113, 122
контроль со стороны ВМ 243, 255
условное 122
- Визуализация микросхем 446
- Виртуальная машина
API 226, 264
push / pop 211
ветвление 243
вызовы и возврат функций 246
поддержка операционной системы 245
принципы 205, 239
реализация 218, 243
сегменты виртуальной памяти 215
спецификация 243, 255
стек 211
стековая арифметика 213
эмулятор ВМ 225
- Виртуальная машина Java (JVM) 206, 209, 236
- Виртуальные регистры 130, 136
- Внутренние контакты 47
- Встроенные микросхемы (чипы)
HDL 440
методы встроенных микросхем 468
обзор 61
переменные встроенных микросхем 467
- Высокоуровневое программирование
(на языке Jack)
вызовы процедур 294
выражения 292
высказывания 292
использование операционной системы 283

- написание приложений 296
- переменные 290
- примеры программ. См. Примеры программ на языке Jack
- приоритет операторов 292
- создание и удаление объектов 295
- стандартная библиотека классов 278
- строки 288
- структура программ 283
- типы данных 286
- Гарвардская архитектура 169
- Генерация кода. См. Компиляция (генерация кода)
- Глобальный стек 251
- Гонка данных 104, 446
- Грамматика 306, 308, 309, 316
- Графический интерфейс микросхем 446
- Графический процессор (GPU) 170, 395
- Двоичные числа 68
- Декларативность языка HDL 435
- Деление столбиком 388
- Демультимплексор 35, 53, 57
- Дерево деривации 311
- Дерево синтаксического анализа. См. Дерево деривации
- Дизъюнктивная нормальная форма (ДНФ) 428
- Загруженный код 263, 269, 270
- И-вентиль. См. AND-вентиль
- ИЛИ-вентиль 52
- Инвертор 51
- Информационное голодание 149
- ИСКЛЮЧАЮЩЕЕ ИЛИ, вентиль. См. XOR-вентиль
- Карта памяти
 - клавиатуры 133, 157
 - концепция 130, 150
 - экрана 132, 156, 300
- Карта памяти клавиатуры 130, 156, 448
- Карта памяти экрана 132, 156
- Кинг-Нюэль, Августа Ада 136
- Команды goto
 - в языке ассемблера 118
 - в языке ВМ 243
- Команды push / pop 217, 265
- Комбинационные микросхемы 87, 95, 442
- Компилятор
 - архитектура программного обеспечения 369
 - генерация кода 335
 - использование операционной системы 345
 - обработка текущего объекта 348, 351
 - объектно-ориентированные языки 200
 - синтаксический анализ 303
 - спецификация 361
 - таблица символов 334, 338
 - тестирование 377
- Компиляция
 - использование операционной системы 367
- Компиляция (генерация кода)
 - выражений 341
 - высказываний 345
 - конструкторов 349, 353
 - массивов 359
 - методов 354
 - объектов 348

- переменных 337
- процедур 366
- строк 344
- Компьютер Hack
 - архитектура 158, 164
 - обзор 152
 - память данных 146
 - ЦПУ 148, 153
- Компьютер с хранимыми программами (свойство Hack) 423
- Контроль программы. См. Проекты; виртуальная машина, контроль программы
- Лексикон 307
- Логические вентили (См. также отдельные вентили)
 - абстракция 40
 - примитивные и составные 42
 - реализация 44
- Локальная переменная 215, 249, 338
- Макрокоманды 142, 191
- Машина Тьюринга 145
- Машинные команды (инструкции)
 - выбор 162
 - выборка — исполнение 149
 - выполнение 161
 - декодирование 161
 - память 147
 - регистры 148
- Машинный язык
 - адресация 121
 - ввод / вывод 131
 - ветвление 121
 - обзор 113
 - память 119
 - переменные 122
 - пример 124
 - регистры 120
 - символы 129
 - синтаксические условности и форматы файлов 133
 - спецификация 126
- Машинный язык Hack, спецификация 177
- Метаязык 309
- Метод дополнения до двух 385
- Микросхемы
 - визуализация 446
 - встроенные микросхемы 61, 440
 - комбинационные 87, 95, 442
 - порядок реализации 450
 - последовательностные 87, 95, 443, 444
 - с графическим интерфейсом 446
 - синхронизируемые. См. Микросхемы, последовательностные
 - тактируемые. См. Микросхемы, последовательностные
- Мнемоники 174
- Моделирование поведения 58
- Модульный дизайн 29
- Мультиплексор 30, 35, 52
- Наиболее значимые (самые старшие) биты 71
- Наименее значимые (самый младшие) биты 71
- НЕ-вентиль. См. NOT-вентиль
- Обработка стека. См. Программы; ВМ-транслятор, обработка стека (проект 7)
- Объектно-ориентированные языки 301, 348

- Объектные переменные 200, 348
- Оперативное запоминающее устройство (ОЗУ) 27, 33, 89, 99
- в операционной системе 408
- компьютера Nask 486
- микросхемы 30
- описание 29
- происхождение термина 146
- регистры 98
- секвенциальная логика 95
- Операционная система 381
- peek и poke 395
- алгебраические алгоритмы 385
- ввод с клавиатуры 401
- вывод символов 391, 399
- геометрические алгоритмы 397
- обработка строк 391
- основы 383
- управление кучей. См. Операционная система, управление памятью
- управление памятью 392
- эффективность 385
- Операционная система Jack
- API 481
- класс Array 483
- класс Keyboard 485
- класс Math 481
- класс Memory 486
- класс Output 483
- класс Screen 484
- класс String 482
- класс Sys 486
- реализация 404
- спецификация 403
- Оптимизация 424
- Память
- и время 90
- обзор 87
- реализация 102
- триггеры 93, 96
- Память данных 120, 146, 152
- Парсер 183, 227
- Парсинг. См. Синтаксический анализ
- Переключающие устройства 25
- Переменные параметров 290
- Переменные экземпляров 280
- Переполнение стека 254
- Последним пришел — первым вышел (LIFO) 211, 249
- Последовательностные микросхемы 87, 95, 442
- Постоянное запоминающее устройство (ПЗУ) 168
- Построение булевых функций 427
- выразительная сила NAND 429
- дизъюнктивная нормальная форма (ДНФ) 428
- Предопределенные символы 129, 179
- Примеры программ на машинном языке 111
- адресация (указатели) 138
- символьная и двоичная запись 176
- сложение 135
- Примеры программ на языке Jack 277
- Hello World 277
- итеративная обработка 278
- обработка массивов 278
- пример объектно-ориентированного программирования с одним классом 279

- простая компьютерная игра (Square) 297, 300
- реализация связанного списка как пример объектно-ориентированного программирования с множественностью классов 281
- Принцип хранимой программы 144
- Проекты
 - ассемблер (проект 6) 188
 - булева арифметика (проект 2) 84
 - булева логика (проект 1) 62
 - виртуальная машина, контроль программы (проект 8) 266
 - ВМ-транслятор, обработка стека (проект 7) 230
 - высокоуровневый язык (проект 9) 299
 - компилятор, синтаксический анализатор (проект 10) 325
 - компиляция, генерация кода (проект 11) 374
 - компьютерная архитектура (проект 5) 165
 - машинный язык (проект 4) 138
 - операционная система (проект 12) 413
 - память (проект 3) 107
- Промежуточный код 205
- Процедурные языки 354
- Псевдоинструкции 134, 177
- Размер слова 70
- Растровый редактор изображений 300
- Регистры 29, 98
- Регистры данных 114, 149
- Рисование круга 398
- Рисование линии 397
- Рисование пикселя 396
- Сегменты виртуальной памяти ВМ 215
- Секвенциальная логика 90
- Символы (машинного языка) 129
- Символы меток 131, 179
- Симулятор аппаратуры 45, 58
- Синтаксический анализ 303
 - грамматика 306, 308
 - дерево деривации 311
 - дерево синтаксического анализа 311
 - лексический анализ 307
 - метаязык 309
 - парсер 312
 - парсинг 312
 - реализация 320
 - синтаксический анализатор 318
 - синтаксический разбор с рекурсивным спуском 313
 - таблица символов 320
 - токенизатор Jack 320
- Синтаксический разбор с рекурсивным спуском 313
- Система поддержки исполнения программ 240
- Создание и удаление объектов (в языке Jack) 295
- Спецификация языка Nask. См. Язык ассемблера Nask, спецификация
- Стандартная библиотека классов 195, 200, 382. См. также Операционная система
- Статические переменные 200, 290, 338
- Стековая машина 210
- Сумматор
 - абстракция 74
 - реализация 82

- Счетчик команд 149, 152, 162
- Таблица истинности 38, 39
- Таблица символов
- ассемблера 173, 181, 187
 - компилятора 320
 - при генерации кода 334, 338
- Тактовые циклы 463
- Тестовые сценарии 48, 451, 471
- Токенизатор 307
- Триггер 89, 93, 97
- Тьюринг, Алан 207
- Указатель 137, 348
- Упрощение булевых выражений 426
- Ускоренный перенос 85
- Условное ветвление 122
- фон Нейман, Джон 119
- Хеш-таблица 187
- Центральное процессорное устройство (ЦПУ) 148, 161
- абстракция 153
 - булева арифметика 67
 - и архитектура фон Неймана 145
 - компьютера Наск 153
 - машинный язык 113
 - память 109
- Цепочка вызовов 248
- Цикл выборки — исполнения 149
- ЦПУ 153. См. Центральное процессорное устройство (ЦПУ)
- память 164
 - реализация 161
- Черча — Тьюринга гипотеза (тезис) 27
- Чипы. См. Микросхемы
- Шеннон, Клод 41
- Шины 438, 454
- Шрифты 384, 400
- Эмулятор ЦПУ 139
- тестирование программ на машинном языке 471
- Язык ассемблера Наск, спецификация 177
- Языки ассемблера 115
- Язык описания аппаратуры (HDL)
- визуализация микросхем 446
 - встроенные микросхемы 440
 - единица времени 463
 - контуры обратной связи 446
 - многобитные шины 438
 - множественные выходы 455
 - несоединенные контакты 453
 - нумерация битов и синтаксис шины 438
 - основы 433
 - последовательностные составные микросхемы 445
 - практический справочник по HDL 449
 - пример программы 434
 - синтаксические ошибки 452
 - создание шин из внутренних контактов (индексирование) 454
 - структура программы 436
 - файлы и тестовые сценарии 451
- Язык описания тестов 457
- обзор 457
 - тестирование микросхем 461
 - тестирование программ ВМ 473
 - тестирование программ на машинном языке 471
- Язык программирования Jack. См. Высокоуровневое программирование на языке Jack

ЛУЧШИЕ КНИГИ О БИЗНЕСЕ С ЛОГОТИПОМ ВАШЕЙ КОМПАНИИ? ЛЕГКО!

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

ПАРТНЕРСКИЕ ОПЦИИ:

- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

ПАРТНЕРСКИЕ ВОЗМОЖНОСТИ:

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).



У вас есть возможность обсудить свои пожелания с менеджерами корпоративных продаж. Как?

Звоните:

+7 495 411 68 59, доб. 2261

Заходите на сайт:

eksmo.ru/b2b



Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Издание для досуга

КЛАССИКА ИТ. ГЛАВНЫЕ КНИГИ ДЛЯ ПРОГРАММИСТОВ

Нисан Ноам
Шокен Шимон

АРХИТЕКТУРА КОМПЬЮТЕРНЫХ СИСТЕМ
КАК СОБРАТЬ СОВРЕМЕННЫЙ КОМПЬЮТЕР ПО ВСЕМ ПРАВИЛАМ

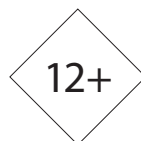
Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Литературный редактор *Н. Бабаева*
Ответственный редактор *Д. Калачева*
Младший редактор *Д. Данилова*
Художественный редактор *Е. Анисина*
Компьютерная верстка *Э. Брегис*
Корректоры *Ю. Никитенко, Л. Макарова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

В оформлении обложки использована фотография:
Yurchanka Siarhei / Shutterstock.com
Используется по лицензии от Shutterstock.com

ООО «Издательство «ЭКСМО»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 кабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Тауар белгісі: «ЭКСМО»
Интернет-магазин: www.book24.ru
Интернет-магазин: www.book24.kz
Интернет-дүкен: www.book24.kz
Импортер в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибутор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибутор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС.
Алматы қ., Домбровский көш., 3-а, литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification
Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «ЭКСМО»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 15.06.2023.
Формат 70х100^{1/16}. Печать офсетная. Усл. печ. л. 40,19.
Тираж экз. Заказ



Москва. ООО «Торговый Дом «Эксмо»

Адрес: 123308, г. Москва, ул. Зорге, д. 1, строение 1.

Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями обращаться в отдел зарубежных продаж ТД «Эксмо»

E-mail: international@eksmo-sale.ru

International Sales: International wholesale customers should contact Foreign Sales Department of Trading House «Eksmo» for their orders.

international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении, обращаться по тел.: +7 (495) 411-68-59, доб. 2151.

E-mail: borodkin.da@eksmo.ru

Оптовая торговля бумажно-беловыми

и канцелярскими товарами для школы и офиса «Канц-Эксмо»:

Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2, Белокаменная ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).

e-mail: kanc@eksmo-sale.ru, сайт: www.kanc-eksmo.ru

Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде

Адрес: 603094, г. Нижний Новгород, улица Карпинского, д. 29, бизнес-парк «Грин Плаза»
Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail:** reception@eksmonn.ru

Филиал ООО «Издательство «Эксмо» в г. Санкт-Петербурге

Адрес: 192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 84, лит. «Е»
Телефон: +7 (812) 365-46-03 / 04. **E-mail:** server@szko.ru

Филиал ООО «Издательство «Эксмо» в г. Екатеринбурге

Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2щ
Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08)

Филиал ООО «Издательство «Эксмо» в г. Самаре

Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»
Телефон: +7 (846) 207-55-50. **E-mail:** RDC-samara@mail.ru

Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону

Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, 44А
Телефон: +7(863) 303-62-10. **E-mail:** info@rnd.eksmo.ru

Филиал ООО «Издательство «Эксмо» в г. Новосибирске

Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3
Телефон: +7(383) 289-91-42. **E-mail:** eksmo-nsk@yandex.ru

Обособленное подразделение в г. Хабаровске

Фактический адрес: 680000, г. Хабаровск, ул. Фрунзе, 22, оф. 703
Почтовый адрес: 680020, г. Хабаровск, А/Я 1006
Телефон: (4212) 910-120, 910-211. **E-mail:** eksmo-khv@mail.ru

Республика Беларусь: ООО «ЭКСМО АСТ Си энд Си»

Центр оптово-розничных продаж Cash&Carry в г. Минске

Адрес: 220014, Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto»

Телефон: +375 17 251-40-23; +375 44 581-81-92

Режим работы: с 10.00 до 22.00. **E-mail:** exmoast@yandex.by

Казахстан: «РДЦ Алматы»

Адрес: 050039, г. Алматы, ул. Домбровского, 3А

Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99). **E-mail:** RDC-Almaty@eksmo.kz

Полный ассортимент продукции ООО «Издательство «Эксмо» можно приобрести в книжных магазинах «Читай-город» и заказать в интернет-магазине: www.chitai-gorod.ru.

Телефон единой справочной службы: 8 (800) 444-8-444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»

www.eksmo.ru

Розничная продажа книг с доставкой по всему миру.

Тел.: +7 (495) 745-89-14. **E-mail: imarket@eksmo-sale.ru**

В электронном виде книги издательства вы можете
купить на www.litres.ru

ЛитРес:
один клик до книг



 **ЧИТАЙ
ГОРОД**



eksmo.ru

Официальный
интернет-магазин
издательства «Эксмо»



Хочешь стать
автором «Эксмо»?

 **БОМБОРА**
издательство

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать
мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

 bombora.ru  [bomborabooks](https://twitter.com/bomborabooks)   [bombora](https://www.instagram.com/bombora)

Лучший способ понять, как работают компьютеры, — это построить один из них с нуля!

Так считают авторы этой книги и потому предлагают практический подход к изучению компьютерных систем. Внутри вас ждет не только исчерпывающее теоретическое описание работы современного компьютера, но и алгоритм конкретных шагов, необходимых для его конструирования.

БЛАГОДАРЯ КНИГЕ ВЫ НАУЧИТЕСЬ РАЗБИРАТЬСЯ В СЛЕДУЮЩИХ ТЕМАХ:

- Аппаратное обеспечение
- Программирование
- Компьютерная архитектура
- Операционные системы
- Языки низкого и высокого уровня
- Структуры данных и алгоритмы
- Виртуальные машины
- Программная инженерия

Уникальная особенность книги заключается в том, что все эти темы тесно связаны и четко ориентированы на главную цель: создание современной компьютерной системы с нуля.

Учебник полностью самодостаточен: все необходимые знания для построения описанных в нем аппаратных и программных систем есть внутри. Часть I «Аппаратное обеспечение» не требует предварительных знаний, что делает проекты 1–6 доступными для любого студента и самоучки.

Часть II «Программное обеспечение» и проекты 7–12 требуют предварительного изучения программирования (на любом языке высокого уровня).

В отличие от других учебников, которые охватывают только один аспект темы, «Архитектура компьютерных систем» дает целостное и исчерпывающее знание прикладной информатики, необходимое для создания собственных проектов.

ISBN 978-5-04-181053-5



9 785041 810535 >

 **БОМБОРА**
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

 [bombora.ru](https://t.me/bombora.ru)  [bomborabooks](https://twitter.com/bomborabooks)  [bombora](https://facebook.com/bombora)