С. Л. Сергеев

Архитектуры вычислительных систем

bhv®

С. Л. Сергеев

Архитектуры вычислительных систем

Допущено УМО по классическому университетскому образованию
в качестве учебника для студентов высших учебных заведений,
обучающихся по направлению ВПО 010400
«Информационные технологии»

Санкт-Петербург
«БХВ-Петербург»
2010

УДК 681.3.06
ББК 32.973.26-018.2
С32

Сергеев С. Л.

С32 Архитектуры вычислительных систем: учебник. — СПб.: БХВ-Петербург, 2010. — 240 с.: ил. — (Учебная литература для вузов)

ISBN 978-5-9775-0575-8

В учебнике рассмотрена архитектура компьютера на уровне системы команд и адресов. Изложение опирается на минимальное понимание работы "железа" и операционных систем, от читателя требуется лишь знание четырех действий арифметики. Описаны представление данных, диапазон и точность, системы счисления, коды чисел, разновидности команд передачи управления, структура циклов, методы организации переменных адресов. Подробно рассмотрены структура подпрограмм, организация вызова и возврата, методы передачи параметров и сохранения регистров и соответствующие им команды. Описаны конвейер команд и связанные с ним проблемы. Представлены современные направления развития архитектур: RISC- и CISC-процессоры, архитектуры со словом сверхбольшой длины.

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Фото	*Кирилла Сергеева*
Зав. производством	*Николай Тверских*

Рецензенты:

Е. И. Веремей, д. ф.-м. н., профессор, заведующий кафедрой компьютерных технологий и систем факультета прикладной математики — процессов управления Санкт-Петербургского государственного университета

В. А. Кузнецов, д. т. н., профессор кафедры прикладной математики и кибернетики Петрозаводского государственного университета

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.06.10.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 19,35.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0575-8

Оглавление

Введение **7**

Глава 1. Представление данных в компьютере **11**

1.1. Системы счисления 11
1.1.1. P-ичная система счисления 11
1.1.2. Правило перевода из одной системы счисления в другую 13
Целые 13
Числа, меньшие единицы 15
Общее правило перевода 16
1.1.3. Двоичная и вспомогательные системы 18
Примеры перевода из двоичной системы 21
Примеры перевода в двоичную систему 22
1.1.4. Другие системы счисления 23
1.2. Представление двоичных чисел 26
1.2.1. Целые 26
Беззнаковые числа 26
Прямой код 26
Обратный код 27
Дополнительный код 28
Смещенный код 29
1.2.2. Дробные числа 31
Числа с фиксированной точкой 31
Числа с плавающей точкой 32
1.2.3. Диапазон и точность 34
Максимальное и минимальное числа 35
Абсолютная погрешность 36
Относительная погрешность 38
1.3. Представление текстов 40
1.3.1. Кодирование символов 40
1.3.2. Кодирование десятичных чисел 41
1.3.3. Битовые строки 42

Глава 2. Компьютерные вычисления 45

2.1. Операции с битовыми строками 45
2.1.1. Логические сдвиги 45
2.1.2. Операции математической логики 47
2.1.3. Маски 50
2.2. Арифметика целых 52
2.2.1. Операции с беззнаковыми числами 53
2.2.2. Сложение и вычитание целых в прямом коде. Сравнение и признаки результата 55
2.2.3. Сложение и вычитание в дополнительном коде 57
2.2.4. Умножение и деление целых чисел 63
2.2.5. Арифметический сдвиг 65
2.3. Арифметика с плавающей точкой 66
2.3.1. Сложение и вычитание 67
2.3.2. Умножение 70
2.3.3. Деление 72
2.3.4. Квадратный корень 73
2.4. Десятичная арифметика 74

Глава 3. Команды арифметико-логического типа и адресация 79

3.1. Принципиальная схема компьютера 79
3.1.1. Компьютер в целом 79
3.1.2. Память 81
Ячейка 81
Команда 81
Локальная операция 82
3.1.3. Процессор 84
3.2. Основные этапы выполнения команды арифметического типа 85
3.2.1. Трехадресная машина 86
3.2.2. Двухадресные машины 89
Двухадресные машины первого типа 89
Двухадресные машины второго типа 92
3.2.3. Одноадресные машины 96
3.2.4. Сравнение машин разной адресности 98
3.3. Машины с регистрами общего назначения 100
3.3.1. Система команд фиксированной длины 101
3.3.2. Система команд разной длины. Байтовая память 102
3.4. Косвенные, непосредственные и относительные адреса 106
3.4.1. Косвенный адрес 106
3.4.2. Непосредственный адрес 108
3.4.3. Использование регистрового и непосредственного адресов для формирования адресов памяти 111
4.4.4. Относительный адрес 112
3.5. Пересылки 113
3.5.1. Обмен с внешней памятью 115

Глава 4. Команды передачи управления и циклы 117

4.1. Переходы 117
4.1.1. Разветвления в алгоритмах и программах 117
4.1.2. Безусловные переходы 119
4.1.3. Условные переходы. Признаки результата 120
4.1.4. Безусловные и условные переходы по смещению 123
4.2. Циклы 126
4.2.1. Классификация циклов 126
Цикл с заданным числом повторений 128
Цикл итерационного типа 129
Цикл смешанного типа 130
Кратный цикл 131
4.1.2. Переадресация 132
Переадресация с помощью констант, восстановление 132
Косвенные адреса 135
Автоинкремент/декремент 137
Стек 138
Индексный регистр 141
4.2.3. Сложные команды управления циклом 145
Команда управления + продвижение индекса 145
Команда управления + счетчик 145
Команда управления + индексирование + счетчик 146

Глава 5. Подпрограммы и ввод/вывод 149

5.1. Подпрограммы 149
5.1.1. Схема взаимодействия ПП с главной программой 149
5.1.2. Вызов ПП и возврат 152
Засылка в ПП команды возврата 153
Сохранение адреса возврата в регистре 155
Использование стека 158
5.1.3. Передача параметров 160
Стандартные ячейки или регистры 160
Передача параметров через косвенный адрес 161
Передача параметров через стек 163
5.1.4. Сохранение регистров 163
Сохранение регистров в стеке 164
5.1.5. Настройка по месту 167
5.2. Операции ввода/вывода 169
5.2.1. Программно управляемый ввод/вывод 169
5.2.2. Ввод/вывод по прерываниям 171
Прерывания 171
Обработчик прерывания и контроллер 173
5.2.3. Прямой доступ к памяти 174

Глава 6. Параллельность работы и иерархия памяти.....175

6.1. Основные идеи.....175
6.1.1. Иерархия памяти. Идея.....175
6.1.2. Параллельность работы. Идея.....178
6.1.3. Технология взаимодействия уровней памяти.....179
6.2. Виртуальная память.....182
6.2.1. Диск.....183
6.2.2. Страничная организация памяти.....186
Анализ страничной организации.....190
Буфер быстрого преобразования адреса.....192
6.2.3. Сегментная организация.....193
6.2.4. Выводы по использованию виртуальной памяти.....197
6.3. Кэш-память.....197
6.3.1. Кэш прямого отображения.....198
Чтение из кэша.....202
Запись в кэш.....204
Секторированный кэш.....206
6.3.2. Ассоциативный кэш.....207
6.3.3. Множественно-ассоциативный кэш.....208

Глава 7. Организация процессора.....211

7.1. Конвейер команд.....211
7.1.1. Организация конвейера.....211
7.1.2. Задержки конвейера.....213
Задержка работы устройств.....214
Конфликты по ресурсам.....216
Явный конфликт по данным.....218
Скрытые конфликты по данным.....220
7.1.3. Передача управления.....221
Безусловный переход.....222
Условный переход.....223
7.2. Основные направления развития систем команд.....226
7.1.1. RISC-процессоры.....226
7.1.2. CISC-процессоры.....228
Суперконвейер.....229
Суперскалярный конвейер.....229
RISC-ядро.....230
7.1.3. Архитектуры с командным словом сверхбольшой длины.....231

Список литературы.....233

Предметный указатель.....235

Введение

Архитектура ЭВМ — понятие обширное и расплывчатое. В различных разделах компьютерных наук (архитектуры ЭВМ, базы данных, операционные системы) принято описывать объект или технологию на двух уровнях: физическом и логическом. На физическом уровне объект описывается так, как он видится разработчику, на логическом — так, как он видится пользователю. Современный компьютер — это электронное ядро ("железо"), окруженное большим числом различных оболочек. Система команд, транслятор, операционная система — некоторые из них. И у каждой оболочки есть свой разработчик и свой пользователь. Разработчикам этих оболочек и их пользователям компьютер видится по-разному. Таким образом, вместо двух уровней рассмотрения появляется несколько. Иногда, например, предлагают разделить рассмотрение компьютера на шесть уровней (не считая "железа"): цифровой логический, микроархитектурный, архитектуры команд, операционной системы, языка ассемблера, языка высокого уровня.

В данной книге рассматривается уровень системы команд и адресов. Разумеется, деление на уровни условно. Кроме того, автор стремился к тому, чтобы читателям были понятны причины, породившие ту или иную деталь архитектуры системы команд и адресов. Это потребовало местами спускаться на более низкий уровень, разделяя исполнение команд на отдельные этапы, а местами подниматься на более высокий уровень — уровень операционной системы, рассматривая команды ввода/вывода и виртуальную память. И все же, изложение сосредоточено в области системы команд и опирается лишь на минимальное понимание работы "железа" и операционных систем, которое можно получить целиком из материала книги. От читателя не требуется никаких предварительных знаний, кроме четырех действий арифметики.

Еще раз подчеркнем: главная задача книги — не просто рассказать об устройстве компьютера, но и объяснить, почему он так устроен.

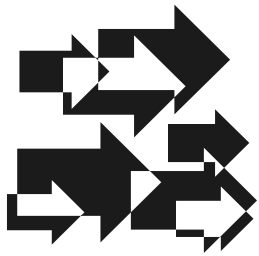
Книга состоит из семи глав.

- *Глава 1 "Представление данных в компьютере"*. В ней рассматриваются различные системы счисления, главным образом, двоичная, восьмеричная и шестнадцатеричная. Даются и обосновываются правила перевода чисел из одной системы счисления в другую. Описываются различные коды чисел, используемые в компьютерах. Показываются преимущества дополнительного и смещенного кодов. Исследуются проблемы диапазона и точности представления чисел. Показывается их взаимозависимость. Рассматриваются способы кодирования символов.
- *Глава 2 "Компьютерные вычисления"*. В ней рассматриваются всевозможные компьютерные операции с кодами чисел и символов (главным образом, операции математической логики, сдвиги и арифметические операции). Цель этой главы — описать все встречающиеся в компьютерах операции арифметико-логического типа (хотя все одновременно они вряд ли встречаются в каком-либо компьютере). Арифметические операции описываются как последовательность отдельных шагов. Это важно для понимания работы конвейера команд.
- *Глава 3 "Команды арифметико-логического типа и адресация"*. Цель этой главы — описать всевозможные модификации команд арифметико-логического типа, имеющиеся в машинах с разной адресностью и с разными способами формирования адресов. Приводится принципиальная схема компьютера в целом и принципиальная схема процессора, описываются основные этапы выполнения команды арифметико-логического типа, подробно рассматривается выполнение команд в машинах разной адресности. Анализируются достоинства машин с регистрами общего назначения и возникающие в них проблемы. Описываются прямой, косвенный, непосредственный и относительный адреса и особенности распределения памяти, связанные с использованием двух последних. Рассматривается также адресация устройств внешней памяти.
- *Глава 4 "Команды передачи управления и циклы"*. В начале главы приводятся разновидности простых команд передачи управления, выполняющих единственное действие — безусловную или условную передачу управления. Далее исследуются циклы — разновидности циклов и их структура. Подробно рассматривается важная разновидность циклов — с переменными командами и со счетчиками. Описывается структура таких циклов и методы организации переменных адресов. Во многих машинах имеются сложные команды управления циклом, совмещающие организацию передачи управления с продвижением индекса и/или со счетчиком. В заключение главы рассматриваются такие команды.
- *Глава 5 "Подпрограммы и ввод/вывод"*. Подпрограммы, наряду с циклами, являются главной структурной единицей программ на нижнем уровне.

В главе подробно рассматриваются проблемы взаимодействия подпрограмм с главной программой: организация вызова и возврата, различные методы передачи параметров и сохранения регистров и соответствующие им команды. Обсуждается проблема настройки подпрограмм по месту и главный прием такой настройки — базирование. Операции ввода/вывода обычно производят обмен сразу большим количеством слов или байтов и потому выполняются как цикл. Обычно они выполняются с помощью подпрограмм, исполняемых процессором или контроллером. Это делает логичным включение операций ввода/вывода в данную главу. В конце главы рассматриваются варианты организации ввода/вывода — программно управляемый, по прерываниям и прямой доступ к памяти.

- *Глава 6 "Параллельность работы и иерархия памяти".* В предыдущих главах показано, что адрес памяти зачастую определяется сложно — комбинацией базирования, индексирования и смещения. В этой главе описываются еще более сложные способы организации оперативной памяти — сегментная и страничная. Обсуждаются достоинства иерархической памяти, особенно ярко проявляющиеся в многозадачном режиме или в однозадачном, при высоких требованиях к объему памяти. Описывается также развитие идеи иерархии памяти — кэш-память. Рассматриваются разновидности кэша, методы (политики) чтения и записи.
- *Глава 7 "Организация процессора".* Описывается конвейер команд. Обсуждаются его достоинства и проблемы, с которыми он сталкивается: задержки работы устройств, конфликты по ресурсам и по данным, проблемы разгона конвейера и предсказания переходов. В конце главы обсуждаются варианты аппаратного решения проблем конвейера — RISC- и CISC-процессоры, архитектуры со словом сверхбольшой длины.

ГЛАВА 1



Представление данных в компьютере

1.1. Системы счисления

1.1.1. P-ичная система счисления

Система счисления — это совокупность правил, позволяющих считать и кратко записывать числа. Подразумевается, что форма записи чисел должна быть удобной для выполнения арифметических операций. В повседневной жизни мы используем десятичную систему счисления (можно также сказать: систему счисления с основанием десять). По аналогии с привычной для нас десятичной системой можно определить и другие системы счисления.

Опишем, например, восьмеричную. Ее базу составляют 8 символов (цифр) — 0, 1, 2, 3, 4, 5, 6, 7. (В десятичной системе базу составляют 10 символов — от 0 до 9.) Любое число в восьмеричной системе изображается только с помощью символов базы. (И точки, разделяющей целую и дробную части числа.)

$$b_n b_{n-1} \ldots b_1 . b_0 b_{-1} \ldots b_{-m}. \qquad (1.1)$$

Выражение (1.1), составленное из символов восьмеричной базы, обозначает число

$$b_n 8^n + b_{n-1} 8^{n-1} + \ldots + b_1 8^1 + b_0 8^0 + b_{-1} 8^{-1} + \ldots + b_{-m} 8^{-m}. \qquad (1.2)$$

Аналогично строятся и другие системы. Например, четверичная система счисления имеет базу из четырех символов: 0, 1, 2, 3. Любое число в ней изображается с помощью только этих символов, имеет вид (1.1) и обозначает

$$b_n 4^n + b_{n-1} 4^{n-1} + \ldots + b_1 4^1 + b_0 4^0 + b_{-1} 4^{-1} + \ldots + b_{-m} 4^{-m}.$$

Рассмотрим, к примеру, выражение $x = 2301.21$. Его можно интерпретировать и как десятичное, и как восьмеричное, и как четверичное число.

Как десятичное:

$$x = 2\times10^3 + 3\times10^2 + 0\times10^1 + 1\times10^0 + 2\times10^{-1} + 1\times10^{-2}.$$

Как восьмеричное:

$$x = 2\times8^3 + 3\times8^2 + 0\times8^1 + 1\times8^0 + 2\times8^{-1} + 1\times8^{-2}.$$

Как четверичное:

$$x = 2\times4^3 + 3\times4^2 + 0\times4^1 + 1\times4^0 + 2\times4^{-1} + 1\times4^{-2}.$$

Очевидно, это совершенно разные числа, хотя и выглядят одинаково. Во избежание путаницы там, где из контекста не ясно, в какой системе счисления записано число, оно снабжается индексом, например,

$$x = 2\times8^3 + 3\times8^2 + 0\times8^1 + 1\times8^0 + 2\times8^{-1} + 1\times8^{-2} = 2301.21_8.$$

Обобщая, можно сказать, что система счисления с основанием p ($2 \le p \le 10$) имеет базу 0, 1, 2, ..., $p-1$. Любое число в этой системе записывается в виде $b_n b_{n-1}\ldots b_1 b_0 . b_{-1}\ldots b_{-m}$, где $0 \le b_i \le p-1$, и интерпретируется как

$$b_n p^n + b_{n-1}p^{n-1} + \ldots + b_1 p^1 + b_0 p^0 + b_{-1}p^{-1} + \ldots + b_{-m}p^{-m}.$$

Надо заметить, что в качестве символов базы не обязательно использовать цифры. Можно использовать любые другие значки. Но все равно эти значки должны быть заменителями слов "ноль", "один" и т. д. Понимание этого позволяет достичь еще большего обобщения.

Пусть система с основанием $p > 0$ (теперь p может быть и больше 10) имеет базу a_0, a_1, ..., a_{p-1} — p различных символов. Тогда любое число, записанное в этой системе $b_n b_{n-1}\ldots b_1 b_0 . b_{-1}\ldots b_{-m}$, где все b_i есть символы из базы, интерпретируется как

$$b_n p^n + b_{n-1}p^{n-1} + \ldots + b_1 p^1 + b_0 p^0 + b_{-1}p^{-1} + \ldots + b_{-m}p^{-m}.$$

Символы базы могут быть произвольны, но обычно используют цифры от 0 до $p-1$, если $p \le 10$, или от 0 до 9 плюс дополнительные символы (если $p > 10$). В частности для шестнадцатеричной системы используют символы: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Правила счета в p-ичной системе такие же, как в десятичной. Надо только помнить, что после старшего символа базы идет число 10. Например, в восьмеричной: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, ..., 17, 20, ..., 27, 30, ..., 77, 100, 101, ...

В шестнадцатеричной: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, ..., 19, 1A, 1B, ..., 1F, 20, ..., 99, 9A, 9B, ..., 9F, A0, A1, ..., FF, 100, 101, ...

В двоичной: 0, 1, 10, 11, 100, 101, 110, 111, 1000, ...

Арифметические операции с числами в p-ичных системах выполняются по тем же правилам, что и в десятичной, только используются свои таблицы сложения и умножения. Пример для $p = 2$ представлен на рис. 1.1.

x \ y	0	1
0	0	1
1	1	10

x \ y	0	1
0	0	0
1	0	1

Рис. 1.1. Таблицы сложения и умножения для $p = 2$

Для $p = 4$ — рис. 1.2.

x \ y	0	1	2	3
0	0	1	2	3
1	1	2	3	10
2	2	3	10	11
3	3	10	11	12

x \ y	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	10	12
3	0	10	12	21

Рис. 1.2. Таблицы сложения и умножения для $p = 4$

1.1.2. Правило перевода из одной системы счисления в другую

Целые

Перевести целое число x в p-ичную систему, значит представить его в виде

$$b_n b_{n-1} \ldots b_1 b_0 ,$$

где b_i — символы из базы p-ичной системы. Для этого надо найти многочлен

$$x = b_n p^n + b_{n-1} p^{n-1} + \ldots + b_1 p + b_0 .$$

Очевидно, можно записать этот многочлен в виде

$$x = x_1 p + b_0,$$

где

$$x_1 = b_n p^{n-1} + b_{n-1} p^{n-2} + \ldots + b_1.$$

Следовательно, b_0 — младшая цифра p-ичного представления числа x и может быть получена как остаток от деления x на p.

Частное — x_1 представим, в свою очередь, как сумму

$$x_1 = x_2 p + b_1.$$

Следовательно, следующая, вторая справа цифра p-ичного представления числа x есть остаток от деления x_1 на p. Продолжая делить каждое новое частное на p, будем получать все новые остатки, являющиеся последовательными (справа налево) цифрами числа x в p-ичной системе. Последнее частное, меньшее p, будет старшей цифрой x.

Пример 1. Перевод числа 1995_{10} в восьмеричную систему (рис. 1.3).

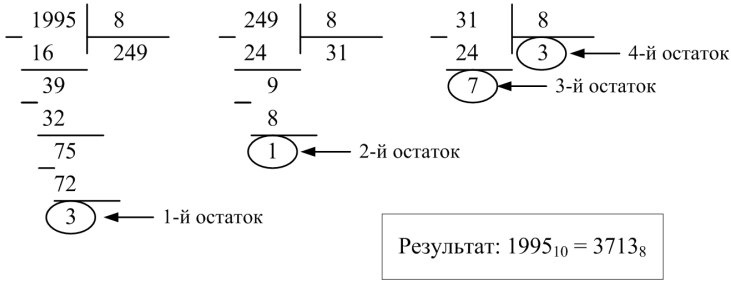


Рис. 1.3

Пример 2. Перевод того же числа в шестнадцатеричную систему (рис. 1.4).

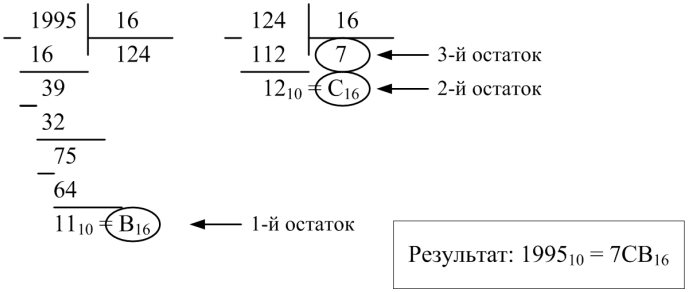


Рис. 1.4

Числа, меньшие единицы

Переведем теперь число y, меньшее 1, из десятичной системы в p-ичную. Будем искать y в виде многочлена

$$y = b_{-1}p^{-1} + b_{-2}p^{-2} + b_{-3}p^{-3} + \ldots + b_{-m}p^{-m},$$

который определяет p-ичную форму числа y: $y_p = 0.b_{-1}b_{-2}b_{-3}\ldots b_{-m}$.

Преобразуем многочлен к виду

$$y = p^{-1}\left(b_{-1} + y_1\right), \text{ где } y_1 = b_{-2}p^{-1} + b_{-3}p^{-2} + \ldots + b_{-m}p^{-m+1}.$$

Из того, что $y_1 < 1$ заключаем, что произведение $y \times p$ имеет целой частью b_{-1} — первую слева цифру p-ичного представления числа y. Умножая y_1 на p, получим число, целая часть которого — вторая цифра y_p, и т. д.

Заметим, что мы еще только ищем представление y в виде многочлена. Заранее нам не известны не только его коэффициенты, но и степень m. Иногда на каком-то шаге оказывается, что $y_i = 0$. Тогда процесс перевода заканчивается $(i = m)$. Чаще этого не происходит слишком долго, поэтому мы должны решить, сколько коэффициентов будем искать. (Иными словами, с какой точностью требуется выполнить перевод в p-ичную систему.)

Пример 3. Перевод десятичного числа 0.1995 в восьмеричную систему (на рис. 1.5, слева) и в шестнадцатеричную систему (рис. 1.5, справа).

0.	1995	0.	1995
	8		16
1.	5960	3.	1920
	8		16
4.	7680	3.	0720
	8		16
6.	1440	1.	1520

Рис. 1.5

Итак, $0.1995_{10} \approx 0.146111_8 \approx 0.33126\ E_{16}$.

Как следует из выведенного нами правила, число, появившееся перед точкой, в последующем умножении не участвует (отделено вертикальной чертой). Ответ получается при прочтении сверху вниз того, что расположено левее вертикальной черты.

Равенства, разумеется, приближенные, т. к. процессы перевода не закончены и младшие цифры восьмеричного и шестнадцатеричного чисел отброшены. Точные переводы получаются редко.

Пример 4. Перевод числа 0.1025 в восьмеричную (рис. 1.6, слева) и в шестнадцатеричную (рис. 1.6, справа) системы счисления.

0.	1025	0.	1025
	8		16
0.	8200	1.	6400
	8		16
6.	5600	10.	2400
	8		16
4.	4800	3.	8400

Рис. 1.6

Итак, $0.1025_{10} \approx 0.064_{8} \approx 0.1A3_{16}$.

Общее правило перевода

Теперь дадим общее правило перевода чисел из десятичной системы в p-ичную.

Число разбивается на две части — целую и дробную (левее точки и правее). Каждая часть переводится в p-ичную систему по своему правилу — для целых и для дробных. Полученные числа являются целой и дробной частями результата. Их объединяют в одно число (слева от точки и справа).

Пример 5. Перевод числа общего вида — 96.96_{10} в восьмеричную систему (рис. 1.7).

Целая часть

96	8	
8	12	8
16	8	1
16	4	
0		

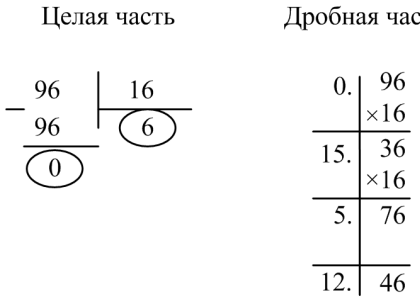
Дробная часть

0.	96
	×8
7.	68
	×8
5.	44

$95_{10} = 140_{8}$ $0.95_{10} \approx 0.75_{8}$ $96.96_{10} \approx 140.75_{8}$

Рис. 1.7

Пример 6. Перевод числа 96.96_{10} в шестнадцатеричную систему (рис. 1.8).



$96_{10} = 60_{16}$ $0.96_{10} \approx 0.F5C_{16}$ $96.96_{10} \approx 60.F5C_{16}$

Рис. 1.8

Для перевода из системы счисления с основанием q в систему с основанием p можно воспользоваться правилом перевода из десятичной системы в p-ичную с единственным уточнением — деление и умножение должны выполняться в q-ичной системе счисления.

В **примере 7** дается перевод числа 140_8 в десятичную систему. $q = 8$, $p = 10$, следовательно, вычисления надо производить в восьмеричной системе (рис. 1.9).

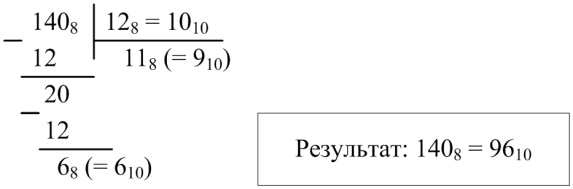


Рис. 1.9

Поскольку такие вычисления очень сложны своей непривычностью, перевод из произвольной системы в десятичную иногда выполняют по более простому правилу. При этом объем вычислений больше, но ведутся они в десятичной системе.

Вот это правило.

От числа $b_n b_{n-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-m}$ переходят к многочлену

$$b_n p^n + b_{n-1} p^{n-1} + \ldots + b_1 p^1 + b_0 p^0 + b_{-1} p^{-1} + b_{-2} p^{-2} + \ldots + b_{-m} p^{-m},$$

значение которого вычисляется в десятичной системе. Например:

$$140.75_8 = 1\times 8^2 + 4\times 8^1 + 0\times 8^0 + 7\times 8^{-1} + 5\times 8^{-2} = 64 + 32 + \frac{7}{8} + \frac{5}{64} \approx 96.959.$$

1.1.3. Двоичная и вспомогательные системы

Для хранения p-ичного n-разрядного числа в памяти компьютера служит ячейка, состоящая из n одинаковых элементов. Каждый элемент — это устройство, способное находиться в одном из p устойчивых состояний. Каждому символу базы ставится в соответствие одно из состояний устройства. Записать символ в элемент ячейки, значит привести этот элемент в соответствующее состояние. Элементы ячейки упорядочены, как и разряды в числе, так что в каждый элемент записывается свой, вполне определенный разряд числа. Элементы ячейки называют также *разрядами*.

Наиболее простые и надежные устройства хранения и переработки информации работают с двоичными числами. Разряды этих устройств — двоичные. Двоичные разряды называют также битами, т. к. в них записывается один бит — минимальная единица информации. Базу двоичной системы составляют символы 0 и 1. В табл. 1.1 приведены некоторые числа в двоичной системе.

Таблица 1.1. Примеры двоичных чисел

Десятичные	0	1	2	3	4	5	6	7
Двоичные	000	001	010	011	100	101	110	111
Десятичные	8	9	10	11	12	13	14	15
Двоичные	1000	1001	1010	1011	1100	1101	1110	1111
Десятичные	16		32		64		1024	
Двоичные	10000		100000		1000000		10000000000	

Из таблицы видно, что двоичные числа значительно длиннее десятичных, но по ним невозможно понять, во сколько раз. Оценим эту величину. Пусть число x состоит из n разрядов в десятичной системе и из m — в двоичной. Это значит, что выполняются неравенства

$$10^{n-1} \le x < 10^n \text{ и } 2^{m-1} \le x < 2^m.$$

Логарифмируя их по основанию 2, получим:

$$(n-1)\times \log_2 10 \le \log_2 x < n\times \log_2 10 \text{ и } m-1 \le \log_2 x < m.$$

$\log_2 x$ удовлетворяет двум неравенствам одновременно, поэтому

$$\max\left[(n-1)\times\log_2 10\ ,\ m-1\right]\le\log_2 x<\min\left[n\times\log_2 10\ ,m\right].$$

Следовательно,

$$(n-1)\times\log_2 10<m\ \text{ и }\ m-1<n\times\log_2 10\ .$$

Отсюда

$$\left(1-\frac{1}{n}\right)\times\log_2 10<\frac{m}{n}<\log_2 10+\frac{1}{n}.$$

При больших n обе границы стремятся к $\log_2 10\approx 3.3$. Таким образом, при больших n двоичные числа длиннее десятичных примерно в 3.3 раза.

Использование двоичной системы счисления добавляет к обычной для недесятичных систем трудности — непривычности, еще одну — очень большую длину чисел. Работа с ними утомительна и увеличивает вероятность ошибок. В то же время преимущества использования двоичной системы в компьютерах столь убедительны, что не позволяют от нее отказаться.

Компромисс найден в использовании одной из вспомогательных систем — восьмеричной или шестнадцатеричной (для каждого типа компьютера выбирается одна из них). Дело в том, что переводы из восьмеричной системы в двоичную и обратно, так же как и из шестнадцатеричной в двоичную и обратно, настолько просты, что могут быть выполнены человеком очень быстро даже в уме. Поэтому используется такая схема: компьютер работает с двоичными числами, а человек читает и записывает их как восьмеричные (или шестнадцатеричные), мысленно осуществляя перевод из одной системы в другую. В этой схеме сохраняется непривычность манипулирования с недесятичными числами, однако длина этих чисел примерно такая же, как и десятичных.

Возьмем произвольное целое двоичное число: $b_n b_{n-1}\ldots b_1 b_0$.

Запишем его в виде многочлена

$$b_n 2^n+b_{n-1}2^{n-1}+\ldots+b_5 2^5+b_4 2^4+b_3 2^3+b_2 2^2+b_1 2^1+b_0 2^0.$$

Рассмотрим сумму трех правых слагаемых: $b_2 2^2+b_1 2^1+b_0 2^0$. Очевидно, что эта сумма лежит в диапазоне от 0 до 7 и может быть заменена одной восьмеричной цифрой. Обозначим ее через c_0.

Возьмем теперь следующие три слагаемых. Вынося за скобки 2^3, замечаем, что сумму в скобках можно заменить одной восьмеричной цифрой. Пусть это будет c_1.

$$b_5 2^5+b_4 2^4+b_3 2^3=\left(b_5 2^2+b_4 2^1+b_3 2^0\right)\times 2^3=c_1\times 8.$$

Следующие три слагаемых будут иметь общим множителем 2^6, а выражение в скобках снова заменим восьмеричным — c_2.

Аналогично будем поступать с каждой следующей тройкой слагаемых. Если число слагаемых в многочлене не кратно трем, добавим слагаемое вида $0\times 2^{n+1}$ или два слагаемых: $0\times 2^{n+2}+0\times 2^{n+1}$ с тем, чтобы последняя группа слагаемых также была тройкой. В результате придем к новому многочлену

$$c_k 8^k + c_{k-1}8^{k-1} + \ldots + c_1 8^1 + c_0 8^0,$$

где k равно $\frac{n}{3}$, $\frac{n+1}{3}$ или $\frac{n+2}{3}$ (выбирается целое).

Но это соответствует восьмеричному числу $c_k c_{k+1}\ldots c_1 c_0$.

Теперь возьмем двоичное число, меньшее единицы $0.b_{-1}b_{-2}b_{-3}\ldots b_{-m}$. Построим соответствующий многочлен:

$$b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \ldots + b_{-m}2^{-m}.$$

Рассмотрим сумму из трех левых слагаемых. Если в ней вынести за скобки 2^{-3}, то в скобках получится сумма, которую можно заменить одной восьмеричной цифрой. Обозначим ее c_{-1}.

$$b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} = \left(b_{-1}2^2 + b_{-2}2^1 + b_{-3}2^0\right)\times 2^{-3} = c_{-1}\times 2^{-3}.$$

В следующих трех слагаемых за скобку вынесем 2^{-6}. Сумму в скобках заменим восьмеричной цифрой c_{-2}.

Аналогично будем действовать с каждой следующей тройкой слагаемых. Если m не кратно трем, добавим слагаемое $0\times 2^{-m-1}$ или два слагаемых: $0\times 2^{-m-1}+0\times 2^{-m-2}$ так, чтобы последняя группа слагаемых также оказалась тройкой. В результате получим новый многочлен: $c_{-1}8^{-1} + c_{-2}8^{-2} + \ldots + c_{-r}8^{-r}$, соответствующий восьмеричному числу $0.c_{-1}c_{-2}\ldots c_{-r}$, где $r = \frac{m}{3}$, $\frac{m+1}{3}$ или $\frac{m+2}{3}$.

Общее правило перевода из двоичной системы в восьмеричную таково:

1. Начиная от десятичной точки, влево и вправо объединяем цифры двоичного числа в тройки (триады), дополняя, при необходимости, число нулями слева и справа.

2. Каждую триаду заменяем соответствующей восьмеричной цифрой по табл. 1.2.

Таблица 1.2. Триады

Двоичная триада	000	001	010	011	100	101	110	111
Восьмеричная цифра	0	1	2	3	4	5	6	7

Правило перевода из восьмеричной системы в двоичную:

1. Каждую восьмеричную цифру заменяем соответствующей триадой (именно триадой, т. е. 0 заменяем на 000, 1 — на 001 и т. д.).
2. Отбрасываем слева и справа лишние нули.

Если в этих двух правилах слова "восьмеричная" заменить на "шестнадцатеричная", "тройки" на "четверки", а "триады" на "тетрады", то получим правила перевода из двоичной системы в шестнадцатеричную и обратно. В табл. 1.3 приведены соответствующие тетрады.

Таблица 1.3. Тетрады

Двоичная тетрада	0000	0001	0010	0011	0100	0101	0110	0111
16-ричная цифра	0	1	2	3	4	5	6	7
Двоичная тетрада	1000	1001	1010	1011	1100	1101	1110	1111
16-ричная цифра	8	9	A	B	C	D	E	F

Примеры перевода из двоичной системы

Рассмотрим перевод числа $x = 1011011101.100111110_2$ в восьмеричную систему.

1. Разбиваем на триады (от точки вправо и влево) — рис. 1.10.
2. Дополняем нулями (рис. 1.11).
3. Заменяем триады восьмеричными цифрами: $x = 1335.474_8$.

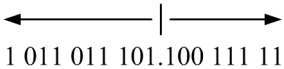


Рис. 1.10

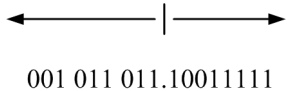
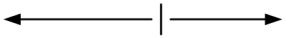


Рис. 1.11

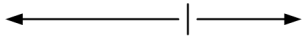
Рассмотрим перевод числа $x = 1011011101.100111110_2$ в шестнадцатеричную систему:

1. Разбиваем на тетрады (от точки вправо и влево) — рис. 1.12.



10 1101 1101.1001 1111

Рис. 1.12



0010 1101 1101.1001 1111

Рис. 1.13

2. Дополняем нулями (рис. 1.13).
3. Заменяем тетрады шестнадцатеричными цифрами: $x = 2DD.9E_{16}$.

Примеры перевода в двоичную систему

Переведем в двоичную систему число $x = 124.124_8$.

1. Заменяем цифры триадами: 001 010 100.001 010 100.
2. Отбрасываем лишние нули: $x = 1010100.\ 0010101_2$.

Переведем в двоичную систему число $x = 124.124_{16}$.

1. Заменяем цифры тетрадами: 0001 0010 0100.0001 0010 0100.
2. Отбрасываем лишние нули: $x = 100100100.\ 000100100 1_2$.

Обычно проблемы перевода из одной системы счисления в другую скрыты от человека, работающего с компьютером: человек передает компьютеру информацию в десятичной системе, компьютер переводит ее в двоичную, обрабатывает в двоичной системе, а перед выдачей результатов человеку переводит их в десятичную систему.

Редко, но все же иногда приходится общаться с компьютером на его языке. При этом человек формулирует входную информацию и получает выходную в восьмеричной или шестнадцатеричной системе. Перевод в двоичную и обратно осуществляется на уровне устройств ввода/вывода: мы нажимаем клавишу, на которой изображена, например, шестнадцатеричная цифра, а клавиатура передает в процессор соответствующую тетраду. И наоборот, из процессора передается двоичный код, а в принтере или на экране монитора каждой тетраде кода соответствует шестнадцатеричная цифра, которая и печатается.

1.1.4. Другие системы счисления

Рассмотренные нами системы счисления называют *системами с неотрицательной базой*, т. к. существуют системы, в которых часть элементов базы отрицательна.

Например, "троичная с симметричной базой". (Имеется в виду симметрия относительно нуля.) В ней элементы базы — 0, 1 и –1 (последнюю цифру обозначают с чертой сверху вместо минуса: $\bar{1}$).

В табл. 1.4 приводятся некоторые числа в этой системе.

Таблица 1.4. Троичная система с симметричной базой

Десятичные	0	1	2	3	4	5	6	7	8	9	10
Троичные	0	1	$1\bar{1}$	10	11	$1\bar{1}\bar{1}$	$1\bar{1}0$	$1\bar{1}1$	$10\bar{1}$	100	101
Десятичные	27		81		243		729				
Троичные	1000		10000		100000		1000000				

Здесь число по-прежнему интерпретируется как многочлен, но коэффициенты могут быть и отрицательными. К примеру, числу $1\bar{1}\bar{1}$ соответствует многочлен $1\times3^2-1\times3^1-1\times3^0=5$, а числу $10\bar{1}$ — многочлен $1\times3^2+0\times3^1-1\times3^0=8$. Важным достоинством троичной системы является единообразие изображения положительных и отрицательных чисел. Здесь не требуется знаков "–" и "+" (табл. 1.5).

Таблица 1.5. Отрицательные числа в троичной системе с симметричной базой

Десятичные	–1	–2	–3	–4	–5	–6	–7
Троичные	$\bar{1}$	$\bar{1}1$	$\bar{1}0$	$\bar{1}\bar{1}$	$\bar{1}11$	$\bar{1}10$	$\bar{1}1\bar{1}$

Просто изменять знак числа: все 1 надо поменять на $\bar{1}$, а $\bar{1}$ на 1. Троичная система даже предлагалась как конкурент двоичной для использования в компьютерах. Двоичная система победила лишь благодаря значительно большей надежности хранения информации в двоичных устройствах.

P-ичной системе счисления не обязательно иметь неотрицательную или симметричную базу. В табл. 1.6 приведены разновидности пятеричных систем счисления (в первом столбце — система с неотрицательной базой, в третьем — с симметричной). Заметим, что только системе с неотрицатель-

ной базой требуется специальный знак для изображения отрицательных чисел.

Таблица 1.6. Различные пятеричные системы

Десятичные числа	База системы			
	0, 1, 2, 3, 4	$\overline{1}$, 0, 1, 2, 3	$\overline{2}$, $\overline{1}$, 0, 1, 2	$\overline{3}$, $\overline{2}$, $\overline{1}$, 0, 1
0	0	0	0	0
1	1	1	1	1
2	2	2	2	$1\overline{3}$
3	3	3	$1\overline{2}$	$1\overline{2}$
4	4	$1\overline{1}$	$1\overline{1}$	$1\overline{1}$
5	10	10	10	10
6	11	11	11	11
7	12	12	12	$1\overline{3}\overline{3}$

Следует подчеркнуть, что основание системы счисления — это вовсе не число символов базы. Система счисления может иметь даже отрицательное основание. Например, система с основанием –2 и базой, состоящей из цифр 0, 1. Числа в этой системе интерпретируются, как и обычно, полиномом:

$$a_n(-2)^n + a_{n-1}(-2)^{n-1} + \ldots + a_1(-2)^1 + a_0(-2)^0.$$

В табл. 1.7 представлены некоторые числа в этой системе счисления.

Таблица 1.7. Система с основанием –2

$p = 10$	0	1	2	3	4	5	6
$p = -2$	0	1	110	111	100	101	11010
$p = 10$	7	8	9	10	11	12	13
$p = -2$	11011	11000	11001	11110	11111	11100	11101

Могут быть системы с иррациональным или даже комплексным основанием.

Все перечисленные системы являются позиционными. В них значение цифры зависит от занимаемой ею позиции в числе. В числе 1991 первая единица

означает одну тысячу, а последняя — одну единицу, первая девятка — девять сотен, а вторая — девять десятков.

Римская система — непозиционная. В ней нет определенной базы и значения цифр не зависят от занимаемого места. Правда, они иногда прибавляются, а иногда вычитаются, так что можно сказать, что в ней присутствует элемент позиционности.

XXXVIII = 10 + 10 + 10 + 5 + 1 + 1 + 1 = 38,

MCMXCI = 1000 – 100 + 1000 – 10 + 100 + 1 = 1991.

Непозиционной является также самая простая из систем счисления — единичная. В ней используется только одна цифра — 1, а значение числа равно количеству единиц: 111 = 3, 111111 = 6.

Есть также целая группа непозиционных систем счисления, которые исследовались на возможность использования в компьютерах. Их называют *системами в остаточных классах* (СОК). В теории чисел доказано, что если m_1, m_2, ..., m_n — попарно взаимно простые числа, то каждому числу x, при $0 \le x < m_1 \times m_2 \times \ldots \times m_n$ взаимно-однозначно соответствует набор остатков от деления x на m_1, m_2, ..., m_n. Пусть, например, $m_1 = 7$, $m_2 = 11$, $m_3 = 13$. $m_1 \times m_2 \times m_3 = 1001$. Любому неотрицательному числу соответствуют три остатка от деления на 7, 11, 13. Причем, если оно не превышает 1000, ни одно другое число из этого диапазона не имеет того же набора остатков. Число 100 в такой системе запишется как 2, 1, 9; число 200 — как 4, 2, 5; число 1000 — как 6, 10, 12.

Интерес к СОК связан с тем, что в этих системах арифметические операции могут выполняться быстрее, чем в позиционных системах. Если, например, в системе с базой m_1, m_2, m_3 два числа представлены остатками x — a_1, a_2, a_3 и y — b_1, b_2, b_3, то числу $z = x + y$ соответствуют остатки c_1, c_2, c_3, где $c_i = a_i + b_i$ или, если $a_i + b_i \ge m_i$, $c_i = a_i + b_i - m_i$. Отсюда видно, что все остатки могут вычисляться независимо, а следовательно, одновременно, что повысит скорость вычислений. Аналогично обстоит дело с другими операциями.

Однако СОК имеют существенные недостатки, среди которых — сложность вычислений с дробными числами, что препятствует их широкому внедрению.

1.2. Представление двоичных чисел

1.2.1. Целые

Будем рассматривать представление двоичных чисел в ячейке памяти, состоящей из n двоичных разрядов. В компьютерах разных типов длина ячеек различна, да и в одном компьютере обычно используются ячейки с разными n. Ячейки с $n = 8$ называются байтами. В примерах будем использовать ячейки с $n = 8, 16, 32$. (Они, по-видимому, наиболее распространены.)

Беззнаковые числа

Ячейки с n двоичными разрядами как раз достаточно для записи n-разрядного целого двоичного числа x (без знака). Так как двоичные числа состоят только из нулей и единиц, наибольшее n-разрядное двоичное число $x_{\max}$ состоит из n единиц (рис. 1.14), а наименьшее — из n нулей.

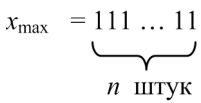


Рис. 1.14

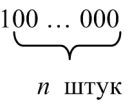


Рис. 1.15

Для быстрого перевода числа, состоящего из n единиц, в десятичную систему достаточно сообразить, что оно на 1 меньше числа, изображенного на рис. 1.15.

Но это число равно 2^n. Таким образом, $x_{\max} = 2^n - 1$. Следовательно, в байте, например, можно записать любое целое без знака от 0 до $2^8 - 1 = 255$, а в двухбайтовой ячейке — от 0 до 65 535.

В ячейках целые числа выравниваются по правому краю. Это выражение означает, что правые границы числа и ячейки совпадают, т. е. разряд единиц числа записывается всегда в крайний правый разряд ячейки. Если число имеет меньше разрядов, чем ячейка, то в левые, свободные разряды ячейки, записываются нули. Например, число 1 записывается в байте как 00000001, число 17_{10} — как 00010001.

Прямой код

При записи в ячейку целых со знаком один разряд приходится отводить для запоминания знака. Обычно знаковым объявляется самый левый разряд ячей-

ки. В знаковый разряд записывается 1, если число отрицательно, 0 — если положительно. Для хранения самого числа остается $n-1$ разряд. Поэтому наибольшее число со знаком имеет вид как на рис. 1.16, что соответствует $2^{n-1}-1$.

$$x_{\max} = \underbrace{011\ \ldots\ 11}_{n-1\ \text{штука}}$$

Рис. 1.16

Для байта это 127, для ячейки из двух байтов — 32 767.

Рассмотренная система кодирования целых со знаком, в которой левый бит отводится под знак, а остальные — под абсолютное значение числа, называется *прямым кодом*. Кроме прямого, в компьютерах используются обратный, дополнительный и смещенный коды.

Обратный код

Положительные числа в *обратном коде* записываются так же, как в прямом. Изменения касаются только отрицательных чисел. Для получения обратного кода отрицательного числа все биты прямого кода (кроме знакового) заменяются на противоположные.

Обратные коды некоторых двухбайтовых чисел приведены в табл. 1.8.

Таблица 1.8. Прямой и обратный коды некоторых чисел

Число	**Прямой код**	**Обратный код**
32 767	0111111111111111	0111111111111111
+5	0000000000000101	0000000000000101
+1	0000000000000001	0000000000000001
–1	1000000000000001	1111111111111110
–5	1000000000000101	1111111111111010
–32767	1111111111111111	1000000000000000

Между прямым и обратным кодами отрицательных чисел имеется соотношение: $|x_{\text{пр}}| + |x_{\text{обр}}| = 2^{n-1} - 1$, где n — длина ячейки в битах.

Если число двухбайтовое, то $|x_{\text{пр}}| + |x_{\text{обр}}| = 111111111111111 = 2^{15} - 1$.

Очевидно, что для получения прямого кода отрицательного числа следует применять ту же процедуру, т. е. обратный код $x_{\text{обр}}$ есть $x_{\text{пр}}$: $(x_{\text{обр}})_{\text{обр}} = x_{\text{пр}}$.

Дополнительный код

Дополнительный код положительных чисел совпадает с их прямым кодом. Для получения дополнительного кода отрицательного числа его сначала переводят из прямого кода в обратный, а затем прибавляют 1. То есть $|x_{\text{доп}}| = |x_{\text{обр}}| + 1$.

Или получают непосредственно из прямого: $|x_{\text{доп}}| = 2^{n-1} - |x_{\text{пр}}|$.

Примеры приведены в табл. 1.9.

Таблица 1.9. Прямой и дополнительный коды

Число	Прямой код	Дополнительный код
32 767	0111111111111111	0111111111111111
+5	0000000000000101	0000000000000101
+1	0000000000000001	0000000000000001
–1	1000000000000001	1111111111111111
–5	1000000000000101	1111111111111011
–32 767	1111111111111111	1000000000000001

Здесь также имеет место формула $(x_{\text{доп}})_{\text{доп}} = x_{\text{пр}}$. С прямым, обратным и дополнительным кодами связана проблема двух нулей. Вот ее суть.

Во многих алгоритмах содержится операция сравнения некоторого результата с нулем. Например, чтобы определить, разделился ли x на y без остатка, надо сравнить остаток с нулем. Так как, по определению, остаток имеет знак делимого, при делении может получиться как остаток +0, так и –0. И хотя +0 = –0, их прямой и обратный коды различны (табл. 1.10).

Таблица 1.10. Прямой и обратный коды нуля

Число	Прямой код	Обратный код
+0	0000000000000000	0000000000000000
–0	1000000000000000	1111111111111111

Поэтому, действуй мы в прямом или в обратном коде, нам придется сравнивать остаток как с одним, так и с другим кодом нуля.

При попытке получить дополнительный код числа –0 из его обратного кода возникнет перенос единицы за пределы ячейки. (Получится число 10000000000000000.) Однако такой результат в дополнительном коде не является переполнением. Выходящая за пределы разрядной сетки единица должна быть просто отброшена. (Это станет ясно после изучения операции сложения в дополнительном коде в *главе 2*.)

Таким образом, дополнительные коды чисел +0 и –0 совпадают:

$$+0 = -0 = 0000000000000000.$$

За счет этого диапазон представимых чисел в дополнительном коде на единицу больше, чем в прямом и обратном (табл. 1.11).

Таблица 1.11. Коды числа –32 768

Число	Прямой код	Обратный код	Дополнительный код
–32 768	Не представимо	Не представимо	1000000000000000

Смещенный код

Недостатком дополнительного кода (как и обратного) является наличие различных правил их формирования для положительных и отрицательных чисел. *Смещенный код* свободен от этого недостатка. Идея смещенного кода заключается в том, чтобы вовсе не иметь знакового бита. Вся ячейка, включая знаковый бит, используется для записи беззнакового кода. Коды положительных и отрицательных чисел образуются по единой формуле:

$$x_{\text{смещ}} = 2^{n-1} + x_{\text{пр}}.$$

В частности для двухбайтовой ячейки $x_{\text{смещ}} = 2^{15} + x_{\text{пр}}$.

В табл. 1.12 приведены примеры чисел в смещенном коде.

Таблица 1.12. Примеры чисел в смещенном коде

Число	Смещенный код
32 767	1111111111111111
+5	1000000000000101
+1	1000000000000001
0	1000000000000000

Таблица 1.12 *(окончание)*

Число	Смещенный код
–1	0111111111111111
–5	0111111111111011
–32 767	0000000000000001

Таким образом, все положительные числа и ноль имеют 1 в левом бите, а отрицательные — 0. Как и в дополнительном коде, здесь нет проблемы двух нулей и представимо число –32 768, не имеющее прямого кода.

Вообще-то рассматривать отдельно знаковый бит не обязательно и в обратном, и в дополнительном кодах. Можно считать, что положительные и отрицательные числа, состоящие из $n-1$ бита, кодируются беззнаковыми числами, состоящими из n бит. $x_{\text{обр}}$ образуется из $\left|x_{\text{обр}}\right|$ приписыванием к числу из $n-1$ бита слева единицы, что можно записать в виде формулы:

$$x_{\text{обр}} = 2^{n-1} + \left|x_{\text{обр}}\right|.$$

Аналогично для дополнительного кода.

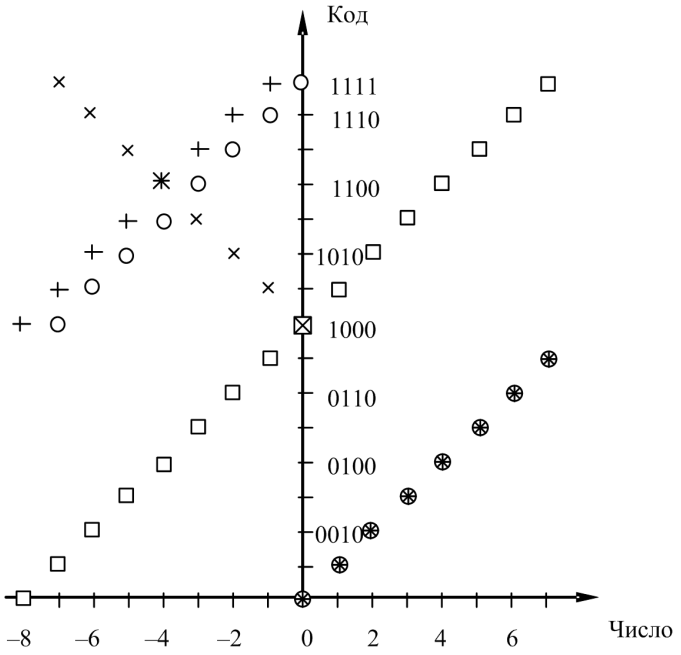


Рис. 1.17. Зависимость кода от числа

Таким образом, формулы перевода из прямого кода в обратный и в дополнительный можно переписать так:

$$x_{\text{обр}} = \begin{cases} x_{\text{пр}}, & \text{при } x_{\text{пр}} \geq 0; \\ 2^n - 1 + x_{\text{пр}}, & \text{при } x_{\text{пр}} < 0. \end{cases} \qquad x_{\text{доп}} = \begin{cases} x_{\text{пр}}, & \text{при } x_{\text{пр}} \geq 0; \\ 2^n + x_{\text{пр}}, & \text{при } x_{\text{пр}} < 0. \end{cases}$$

На рис. 1.17 показана связь между числами в диапазоне от –8 до +7 с их беззнаковыми кодами в четырехбитовой ячейке. (Конечно, четырехбитовых ячеек не бывает, минимум, восьмибитовая, но тогда для рисунка понадобилось бы в 16 раз больше места.)

На рисунке отмечены точки, соответствующие кодам чисел: × — прямым, ○ — обратным, + — дополнительным, □ — смещенным.

1.2.2. Дробные числа

Числа с фиксированной точкой

В общем случае двоичное число имеет как целую, так и дробную части, разделенные точкой. Получается, что для изображения двоичного числа общего вида надо иметь пять различных знаков: для изображения нуля, единицы, знаков "+" и "–", а также точки, разделяющей целую и дробную части. Однако двоичный разряд ячейки способен хранить лишь 0 или 1. Изображая знак числа нулем (если знак — плюс) или единицей (если знак — минус), мы не путаем его с частью самого числа лишь потому, что договариваемся записывать знак в конкретном (обычно самом левом) бите.

Аналогично приходится договариваться о положении точки в числах. Но точка — всегда точка (в отличие от знака, который может быть "+" или "–"), поэтому для нее не нужно отводить бит. Можно записывать подряд целую и дробную части числа, каким-то образом указывая, между какими битами должна стоять точка. Указаний не требуется, если машина работает лишь с числами определенного вида.

Рассмотрим случай, когда используются только числа, по абсолютной величине меньшие единицы. Все они имеют вид $\pm 0.b_{-1}b_{-2}\ldots$, где b_i — какие-либо двоичные цифры. Поэтому можно договориться, что символы "0" и ".", следующие за знаком, не изображаются (эти символы неизменны), а число записывается в память в виде сокращенного кода $\sigma b_{-1}b_{-2}\ldots$, где $\sigma = 0$, если число положительное, и $\sigma = 1$, если отрицательное. Про такие записи говорят, что в них точка фиксирована перед левой цифрой числа.

Если все двоичные числа по абсолютной величине меньше 2, то они имеют вид $\pm b_0.b_{-1}b_{-2}\ldots$ В этом случае можно опускать точку (она всегда стоит после левой цифры числа) и записывать число в память в виде $\sigma b_0 b_{-1}b_{-2}\ldots$

Частным случаем чисел с фиксированной точкой являются целые — у них точка фиксирована после крайней правой цифры.

Существуют машины, работающие только с числами с фиксированной точкой. Такое ограничение не является препятствием к тому, чтобы считать их универсальными, т. к. любые вычисления могут быть сведены к вычислениям с фиксированной точкой.

Пусть, например, требуется вычислить выражение $y = \frac{6.3 \times 0.5 + 2}{128 + 7}$. Его можно привести к виду $y = \frac{0.063 \times 0.5 + 0.02}{0.128 + 0.007}$.

Все числа, входящие в это выражение, и все промежуточные результаты вычислений меньше единицы. Поэтому вычисления могут быть проведены на машине с точкой, фиксированной перед первой цифрой.

Преимущество таких машин — простота вычислений (они ведутся с однотипными числами). Это позволяет делать их более дешевыми. Недостаток — необходимость предварительных преобразований, которые могут оказаться довольно сложными.

Числа с плавающей точкой

Еще один способ записи чисел — с плавающей точкой. В этом случае все числа приводятся к виду $x = M_x \times S^{p_x}$, где M_x — мантисса числа x, p_x — его порядок. Мантисса — число с фиксированной точкой, имеющее тот же знак, что и x, порядок — целое со знаком. S — число, характеризующее данную систему кодирования, одинаковое для всех x в этой системе. Обычно, если x — десятичное, то $S = 10$, если x — двоичное, то $S = 2$ или $S = 16$. В последнем случае можно считать, что x — шестнадцатеричное, т. к. мантиссы x в обеих системах выглядят одинаково.

Например, число $x = 16.125_{10}$ может быть записано в форме с плавающей точкой: $x = 0.16125 \times 10^2$ или $x = 0.016125 \times 10^3$, или $x = 0.00016125 \times 10^5$, или $x = 1612.5 \times 10^{-2}$ и т. д.

Это же число в двоичной системе счисления можно записать так: $x = 10000.001 \times 2^0$ или $x = 1.0000001 \times 2^{100}$, или $x = 0.10000001 \times 2^{101}$, или $x = 0.0010000001 \times 2^{111}$ и т. д. (Здесь, как и в следующем примере, все показатели степени записаны, естественно, в двоичной системе.)

То же число в двоичной системе, но при $S = 16$: $x = 10000.001 \times 16^0$ или $x = 1.0000001 \times 16^1$, или $x = 0.00010000001 \times 16^{10}$ и т. д.

Очевидно, что количество вариантов в каждой системе кодирования неограниченно. Однако не все они годятся для записи в память машины: мантисса в память записывается как число с фиксированной точкой. То есть записывается не сама мантисса, а ее сокращенный код, отличающийся от нее хотя бы отсутствием точки. В большинстве машин должно быть $|M_x| < 1$. В них код отличается от мантиссы еще и отсутствием нуля перед точкой.

В приведенных примерах под требование $|M_x| < 1$ подходят первые три варианта записи чисел в десятичной системе, два последних в двоичной с $S = 2$ и один — с $S = 16$.

Так как S в каждой системе фиксировано, для запоминания числа x необходимо записать в ячейку только два числа: p_x и M_x. Поэтому ячейка делится на 4 части: для знака p_x, для $|p_x|$, для знака M_x, для m_x — сокращенного кода $|M_x|$ (рис. 1.18).

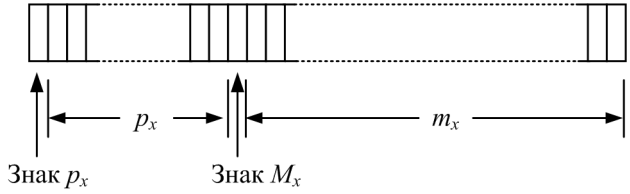


Рис. 1.18. Деление ячейки для представления числа с плавающей точкой

Как размер ячейки, так и деление ее на части для каждой машины свое, фиксированное (программист не может выбирать их по своему усмотрению). В некоторых машинах имеется несколько фиксированных вариантов. Во всех случаях для знаков отводится по одному биту, а для модуля порядка — меньше бит, чем для модуля мантиссы.

Как правило, порядок записывается в коде со смещением, поэтому здесь нет отдельной позиции для знака. p_x называют истинным порядком, а его код, записываемый в память, — Π_x — машинным порядком.

Мантисса записывается в память в прямом коде, поэтому здесь знаковая позиция имеет самостоятельное значение. Ее даже часто отделяют от позиции модуля мантиссы, как показано на рис. 1.19.

Число называется *нормализованным*, если для его изображения используется вариант, при котором порядок имеет наименьшее возможное значение.

Например, в системе с $S=2$ и $|M|<1$ для числа $x=11.011$ из всех возможных вариантов — $x=0.11011\times 2^{10}$, $x=0.011011\times 2^{11}$, $x=0.0011011\times 2^{100}$ и т. д. нормализованным будет первый, для числа $y=0.00101$ нормализованная форма — 0.101×2^{-10}. В память запишутся $m_x=11011$ и $m_y=101$. Формальным признаком нормализованности в этой системе является единица в крайней левой позиции.

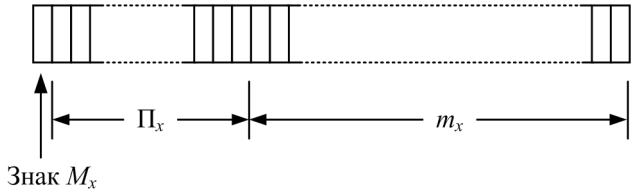


Рис. 1.19. Наиболее распространенное деление ячейки

В системе с $S=2$ и $|M|<2$ нормализованный вид первого числа — 1.1011×2^{1}, второго — 1.01×2^{11}. Машины с $S=2$ и $|M|<2$ обычно работают исключительно с нормализованными числами. Очевидно, в этом случае перед точкой всегда стоит единица, поэтому в сокращенном коде ее опускают. Для таких машин $m_x=1011$ и $m_y=01$.

В системах с $S=16$ и $|M|<1$ в первой позиции нормализованного числа не обязательно находится 1. Здесь левая тетрада должна быть ненулевой. Те же x и y в этих системах выглядят так: $x=11.011=0.0011011\times 16^{1}$, $y=0.00101=0.00101\times 16^{0}$, т. е. $m_x=0011011$, $m_y=00101$.

1.2.3. Диапазон и точность

Диапазон и точность представления в памяти чисел относятся к наиболее важным характеристикам компьютера. Обычно оценка возможностей машины ведется для ее стандартных форматов, т. к. иначе можно говорить о практически неограниченных диапазоне и точности. Будем рассматривать ячейку памяти, состоящую из $n+1$ разряда, один из которых отводится под знак числа. Остальные n разрядов отводятся под модуль числа, если в ячейку записывается целое или число с фиксированной точкой. Для записи в ту же ячейку числа с плавающей точкой n разрядов делятся на две части — k разрядов — для записи порядка, s разрядов — для мантиссы $(k+s=n)$ (рис. 1.20).

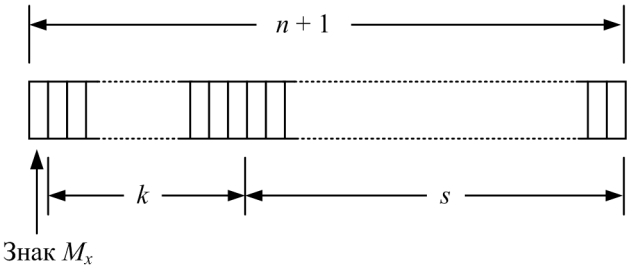


Рис. 1.20. Деление ячейки на части

Рассмотрим следующие характеристики ячейки:

- максимальное представимое число — $x_{\max}$;
- минимальное по абсолютной величине, отличное от нуля число — $x_{\min}$;
- абсолютная погрешность представления числа (ошибка округления) — Δx;
- относительная ошибка — δx.

Найдем эти характеристики для случаев, когда в ячейке хранится:

1. Целое число.
2. Число с фиксированной точкой (при $x < 1$).
3. Число с плавающей точкой (при $|M| < 1$, $S = 2$, $\Pi = p + 2^{k+1}$).
4. Число с плавающей точкой в той же системе кодирования, но с дополнительным условием — мантисса обязательно нормализована ($0 < M < 1$).

Максимальное и минимальное числа

Очевидно, что код максимального числа во всех четырех случаях будет состоять из единиц во всех разрядах, кроме знакового (рис. 1.21 и 1.22).

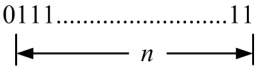


Рис. 1.21. Код максимального числа для первых двух случаев

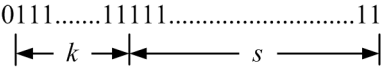


Рис. 1.22. Код максимального числа для двух последних случаев

Минимальное число для первых трех случаев имеет единицу в крайнем правом разряде, в остальных разрядах — нули (рис. 1.23 и 1.24); для четвертого случая единицы должны стоять в крайнем левом разряде мантиссы, в остальных — нули (рис. 1.25).

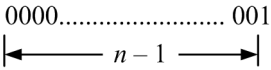


Рис. 1.23. Код минимального числа для первых двух случаев

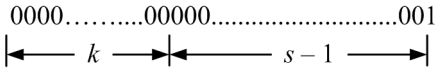


Рис. 1.24. Код минимального числа для третьего случая

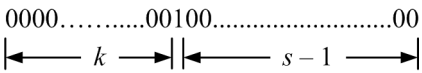


Рис. 1.25. Код минимального числа для последнего случая

Это соответствует числам, показанным в табл. 1.13.

Таблица 1.13. Максимальные и минимальные числа

Формат	$x_{\max}$	$x_{\min}$
Целые	$2^n - 1$	1
С фиксированной точкой	$1 - 2^{-n}$	2^{-n}
С плавающей точкой не нормализованные	$2^{2^{k-1}-1} \times \left(1 - 2^{-s}\right)$	$2^{-2^{k-1}} \times 2^{-s}$
С плавающей точкой нормализованные	$2^{2^{k-1}-1} \times \left(1 - 2^{-s}\right)$	$2^{-2^{k-1}} \times 2^{-1}$

Отбрасывая малые слагаемые, получаем приближенно (табл. 1.14).

Таблица 1.14. Приближенные значения

Формат	$x_{\max}$	$x_{\min}$
Целые	2^n	1
С фиксированной точкой	1	2^{-n}
С плавающей точкой не нормализованные	$2^{2^{k-1}-1}$	$2^{-2^{k-1}-s}$
С плавающей точкой нормализованные	$2^{2^{k-1}-1}$	$2^{-2^{k-1}-1}$

Абсолютная погрешность

Так как ячейка имеет фиксированное количество разрядов для записи модуля числа, любое число из большего количества разрядов может быть представ-

лено в такой ячейке лишь приближенно. Приближенное значение получают или округлением, или просто отбрасыванием лишних разрядов.

При приближенной записи чисел с фиксированной точкой максимальное отбрасываемое число имеет вид, представленный на рис. 1.26.

Это число меньше, чем изображенное на рис. 1.27.

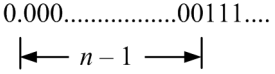


Рис. 1.26. Максимальное отбрасываемое число

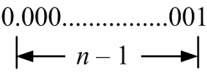


Рис. 1.27. Предел отбрасываемых чисел

А это значит, что максимальное отброшенное число меньше, чем 2^{-n}. Очевидно, что для любых чисел с фиксированной точкой $\Delta x < 2^{-n}$. (Если при записи производится округление, то погрешность еще меньше, т. е. неравенство выполняется тем более.) При записи в ячейку целых чисел мы отбрасываем дробную часть, т. е. погрешность не превышает 1.

При записи чисел с плавающей точкой погрешность мантиссы не превышает, как и для чисел с фиксированной точкой, единицы последнего разряда — 2^{-s}. То есть точное значение мантиссы лежит в пределах $M_{\text{точное}} - 2^{-s} < M_{\text{приближенное}} \le M_{\text{точное}}$.

Умножив неравенство на 2^{p_x}, получим диапазон для приближенного представления самого числа: $x_{\text{точное}} - 2^{p_x - s} < x_{\text{приближенное}} \le x_{\text{точное}}$.

Абсолютная погрешность — это разность между точным значением x и приближенным. Отсюда следует, что $\Delta x < 2^{p_x - s}$. (Конечно, все рассуждения велись для положительных чисел. В общем случае неравенство принимает вид $|\Delta x| < 2^{p_x - s}$.) Таким образом, абсолютные погрешности для трех форматов можно свести в табл. 1.15.

Таблица. 1.15. Абсолютные погрешности

Формат	Δx
Целые	1
С фиксированной точкой	2^{-n}
С плавающей точкой	$2^{p_x - s}$

Из таблицы видно, что в первом случае абсолютная погрешность ни от чего не зависит. Во втором — зависит от размера ячейки: чем больше в ней разрядов, тем меньше погрешность. В формате с плавающей точкой абсолютная погрешность не зависит от значения мантиссы, поэтому она одинакова для нормализованной и ненормализованной мантисс. Она зависит только от длины мантиссы и от порядка числа. Так как $-2^{k-1} \le p_x \le 2^{k-1}-1$, абсолютная погрешность в этом формате лежит в широком диапазоне: $2^{-2^{k-1}-s} \le \Delta x \le 2^{2^{k-1}}$. Диапазоны абсолютных погрешностей сведены в табл. 1.16.

Таблица 1.16. Диапазоны абсолютных погрешностей

Формат	$\min \Delta x$	Δx	$\max \Delta x$
Целые	1	1	1
С фиксированной точкой	2^{-n}	2^{-n}	2^{-n}
С плавающей точкой	$2^{-2^{k-1}-s}$	2^{p_x-s}	$2^{2^{k-1}-s-1}$

Относительная погрешность

В инженерных приложениях имеют дело с *относительной погрешностью*.

Измеряя расстояния между звездами, пренебрегают километрами, измеряя радиус Земли — метрами, размеры мебели — миллиметрами и т. д. То есть объекты разных размеров измеряются в различных единицах (световой год, километр, метр и т. д.). Считается, что измерения двух сильно различающихся по размерам или даже вообще несоизмеримых объектов одинаково точны, если они выполнены с одинаковой относительной погрешностью — $\delta x = \dfrac{\Delta x}{x}$.

С относительной погрешностью связаны выражения типа: "измерить с четырьмя верными знаками" или "вычислить с точностью до пяти верных цифр"; с абсолютной — "измерить с четырьмя верными знаками после запятой" или "вычислить с точностью до пяти верных цифр после запятой". Относительные погрешности целых чисел и чисел с фиксированной точкой зависят от того, какое это число.

Для целых $\delta x = \dfrac{1}{x}$. Поэтому, $\dfrac{1}{x_{\max}} \le \delta x \le \dfrac{1}{x_{\min}}$, т. е. $\dfrac{1}{2^n-1} \le \delta x \le 1$ или $2^{-n} < \delta x \le 1$. Для формата с фиксированной точкой $\delta x = \dfrac{2^{-n}}{x}$. Поэтому $\dfrac{2^{-n}}{1-2^{-n}} \le \delta x \le \dfrac{2^{-n}}{2^{-n}}$ или $2^{-n} < \delta x \le 1$.

В формате с плавающей точкой $\delta x = \dfrac{2^{p_x - s}}{2^{p_x} \times M_x} = \dfrac{2^{-s}}{M_x}$.

Мантиссы нормализованных чисел лежат в диапазоне $\dfrac{1}{2} \le M_x < 1$, поэтому для них $2^{-s} < \delta x \le 2^{-s-1}$. У ненормализованных чисел $2^{-s} \le M_x < 1$, поэтому $2^{-s} < \delta x \le 1$.

Далее приведена табл. 1.17 относительных погрешностей для разных форматов.

Таблица 1.17. Относительные погрешности

Формат	$\min \delta x$	δx	$\max \delta x$
Целые	2^{-n}	$1/x$	1
С фиксированной точкой	2^{-n}	$2^{-n}/x$	1
С плавающей точкой без нормализации	2^{-s}	$2^{-s}/M_x$	1
С плавающей точкой с нормализацией	2^{-s}	$2^{-s}/M_x$	2^{-s+1}

Таким образом, относительная погрешность целых чисел и чисел с фиксированной точкой лежит в одинаковом очень широком диапазоне. Причем, если в левой части этого диапазона относительная погрешность может быть вполне приемлемой, то в правой — совершенно неудовлетворительна. Что касается формата с плавающей точкой без нормализации, то здесь диапазон сужен, причем за счет его левой стороны, т. к. $s < n$, т. е. в целом относительные погрешности хуже.

Если же рассматривать только нормализованные числа, то здесь диапазон относительных погрешностей очень узок. То есть, используя этот формат, мы обеспечиваем практически одинаковую погрешность для всех чисел, которая зависит только от количества знаков мантиссы.

Замечания о выборе формата

- Форматом целых пользуются обычно не для приближенных, а для точных вычислений, когда целое не получено отбрасыванием дробной части, а является целым по своей природе (например, число студентов в аудитории). В этом случае погрешность просто отсутствует.

- Вычисления с фиксированной точкой проще, чем с плавающей, они обеспечивают постоянную низкую абсолютную погрешность. Если $|x| > 2^{-k}$, то $2^{-n} < \delta x < 2^{-s}$, что лучше, чем с плавающей точкой.
- Вычисления с плавающей точкой при нормализованной мантиссе обеспечивают гарантированную достаточно низкую относительную погрешность для любых чисел из очень широкого диапазона.
- Использование ненормализованных чисел целесообразно только в том случае, когда числа выходят из диапазона представимых в нормализованном виде.

1.3. Представление текстов

1.3.1. Кодирование символов

Обработка информации не сводится только к вычислениям. Компьютер должен хранить и перерабатывать нечисловую информацию, записанную буквами и другими символами.

Для размещения символов в двоичной памяти компьютера используются их двоичные коды. Существует много систем кодирования символьной информации. Наиболее распространенными в настоящее время являются различные вариации ASCII-кода (American Standard Code for Information Interchange). В этом коде каждому символу ставится в соответствие определенная комбинация из восьми нулей и единиц — один байт.

Например, код символа A — 01000001, B — 01000010, Z — 01011010. Для строчных букв используются свои коды: a — 01100001, b — 01100010, z — 01111010. Для краткости при записи кодов на бумаге или на экране используют шестнадцатеричную систему. В ней символу A соответствует код 41, B — 42, Z — 5A, a — 61, b — 62, z — 7A. Аналогично кодируются и другие символы, которые могут встретиться в тексте: 2B — код символа "+", 2D — "-", 28 — "(", 29 — ")", 2C — "," , 21 — "!" и т. д.

В тексте могут встретиться и числа. Если не предполагается выполнять с числом каких-либо арифметических операций (например, когда это — номер квартиры или автомобильный номер), то целесообразно не переводить его в двоичную систему, а рассматривать как последовательность символов — составляющих число десятичных цифр.

В ASCII код нуля — 30, единицы — 31 и т. д. Код девятки — 39. Например, число 91 в ASCII кодируется как 0011100100110001; число 723 — как 001101110011001000110011.

Всего символов, используемых в американских текстах, около ста, и для их кодирования достаточно комбинаций из семи нулей и единиц (такими ком-

бинациями можно закодировать 128 различных символов). Первоначально ASCII и состоял из семи битов. Однако потом возникла идея хранить в памяти компьютера в закодированном виде не только тексты, но и таблицы и простые схемы. В связи с этим появилась необходимость запоминать элементы графики (горизонтальные и вертикальные черточки различной толщины, их пересечения и т. д.). Это увеличило количество символов и привело к появлению восьмибитового расширенного ASCII-кода. Восьмибитовый код позволяет записывать 256 различных символов. В настоящее время различные национальные восьмибитовые стандартные коды строятся с использованием ASCII. Младшая половина таблицы кодов сохраняется как в американском стандарте, а старшая (от 128 до 256) используется для представления национальных алфавитов и некоторых символов псевдографики. В некоторых компьютерах используется другой код — EBCDIC (Expanded Binary Coded Decimal Interchange Code). Так же как и в ASCII, каждому символу ставится в соответствие восьмибитовая комбинация или эквивалентная пара шестнадцатеричных цифр.

В обеих системах кодирования старшая часть кода (левая шестнадцатеричная цифра) называется зоной, а младшая — цифрой.

Кодирование символов так же просто, как и перевод чисел из вспомогательных систем счисления — восьмеричной и шестнадцатеричной в двоичную. Да и процесс такой же: используя таблицу, заменяем каждый символ его двоичным кодом. Отличие лишь в том, что при переводе из восьмеричной системы каждая цифра заменяется триадой, при переводе из шестнадцатеричной — тетрадой, а при кодировании символов — байтом. Так что переводы восьмеричных и шестнадцатеричных чисел в двоичную систему и обратно можно называть кодированием восьмеричных и шестнадцатеричных чисел и декодированием.

Как и переводы во вспомогательные системы счисления, кодирование и декодирование символов выполняют устройства ввода и вывода без участия человека или процессора — нажимая клавишу с изображением символа, мы вызываем появление определенной комбинации из восьми битов, приход комбинации из восьми битов в устройство вывода вызывает появление соответствующего символа.

1.3.2. Кодирование десятичных чисел

Как уже говорилось, числа, являющиеся частью текста, рассматриваются как символы и кодируются двумя шестнадцатеричными цифрами. Вместе с тем часто возникает задача хранения больших массивов чисел, внутри которых нет ни одного другого символа. Так как у всех цифр одна и та же зонная часть, ее можно опустить. При этом, правда, нам необходимо помнить, что в

данной части памяти хранятся цифры без зонной части. Такой способ записи десятичных чисел называется *упакованным форматом* или *двоично-десятичным кодом* — BCD (Binary Decimal Code).

Например, число 1995 в ASCII записывается в четырех байтах как 00110001001110010011100100110101 или в краткой, шестнадцатеричной, записи как 31393935. В упакованном формате это же число имеет длину два байта — 0001100110010101. Записывать десятичные числа в двоично-десятичном коде очень просто: нужно каждую цифру перевести в двоичную систему и записать в виде тетрады — совокупности четырех нулей и единиц. Упакованные форматы в ASCII и EBCDIC, естественно, совпадают.

Итак, с вводимым в память компьютера десятичным числом можно поступить трояко:

1. Его можно хранить в памяти как последовательность символов ("распакованный формат" ASCII). В таком виде числа обычно поступают в память.
2. Число можно "упаковать". К этому обычно прибегают, если большое количество чисел расположено подряд.
3. Число можно перевести в двоичную систему. Так поступают с числами, которые будут участвовать в массовых вычислениях.

Примеры приведены в табл. 1.18.

Таблица 1.18. Примеры чисел в разных форматах

Десятичное	Двоичное	BCD	ASCII
7	111	111	00110111
10	1010	10000	0011000100110000
39	100111	111001	0011001100111001
88	1011000	10001000	0011100000111000

1.3.3. Битовые строки

Для каждой системы кодирования характерно свое деление ячейки памяти на поля (части) и интерпретация этих полей.

Записывая в ячейку целое, мы делим ее на два поля: из одного левого бита и из всех остальных. Запись в левом поле интерпретируется как знак числа, в правом — как модуль числа. При записи чисел с плавающей точкой ячейка делится на три поля: поле знака, поле порядка и поле мантиссы. Символы целиком занимают байт (хотя и его, в значительной степени условно, делят на

зонную и цифровую части). В упакованном формате байт делится на два одинаково интерпретируемых поля — в нем записываются два независимых символа.

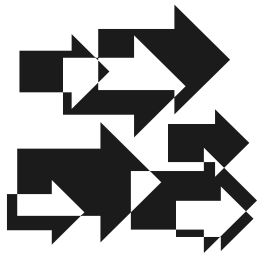
Наряду со стандартными, широко распространенными, используются системы кодирования специальные, частные. От местных или ведомственных стандартов до личных, используемых зачастую в одной-единственной программе. При этом поля могут иметь совершенно произвольные размеры, от одного бита до многих байтов. Цель индивидуальной системы кодирования — экономное использование памяти машины.

Представим себе, например, систему записи анкетных данных. Будем данные выписывать из анкеты подряд всегда в одном и том же порядке, отводя под каждый тип данных всегда одно и то же количество битов. Например, под фамилию — 20 байтов (чтобы наверняка хватило), под год рождения — 1 байт (две последние цифры), под номер телефона — 3,5 байта (7 цифр), под пол — 1 бит (кодируя, например, нулем — женский, единицей — мужской) и т. д.

Другой пример. Представим себе, что отдельные лица отвечают на вопросы некоего теста. Все вопросы требуют ответа "да" или "нет". Кодируя слово "да" цифрой 1, а слово "нет" цифрой 0, результаты опроса каждого лица можно представить как набор цифр 0 и 1. Хотя этот набор будет внешне выглядеть как многозначное двоичное число, на самом деле это именно набор независимых битов. Такие наборы называют битовыми строками.

Понятие битовых строк распространяют и на такие наборы, в которых отдельные последовательности битов объединены в группы, как в предыдущем примере. То есть, вводя любое нестандартное для данной машины деление ячейки на поля, мы вводим битовую строку.

ГЛАВА 2



Компьютерные вычисления

2.1. Операции с битовыми строками

Самая простая из машинных операций — это пересылка, суть которой заключается в копировании содержимого одной ячейки памяти в другую.

Операции с битовыми строками относятся к числу наиболее простых для процессора. По времени выполнения они близки с операцией пересылки.

2.1.1. Логические сдвиги

Пусть ячейка из n битов содержит битовую строку A вида $a_{n-1}a_{n-2}\ldots a_0$.

Операция *"сдвиг влево логический"* (в дальнейшем будем кратко записывать `SSL A` от англ. *Shift Left Logical*) превращает строку A в новую строку — A': $a_{n-2}a_{n-3}...a_0 0$. Происходит последовательный перенос всех битов строки на одну позицию влево. При этом бит, бывший крайним левым, выходит за пределы ячейки и теряется. Справа освобождается один бит, и в него заносится 0. Схематически процесс изображен на рис. 2.1.

Рис. 2.1. Логический сдвиг влево

Если к полученной битовой строке снова применить операцию `SSL`, то получим новую строку: $a_{n-3}a_{n-4}...a_0 00$. k -кратное применение операции приведет к сдвигу исходной строки на k позиций.

В некоторых компьютерах есть операция с параметром `SSL A,k`. В этой команде мы имеем возможность указать параметр `k` — величину сдвига. Если, например, к исходной строке применить операцию `SSL A,2`, то результатом будет $a_{n-3}a_{n-4}...a_0 00..$ Это позволяет нам одной командой выполнить то, что команда простого сдвига `SSL` выполняет за несколько раз. Параметр может принимать любые разумные значения (очевидно, что при $k = n$ битовая строка превратится в нулевую, а сдвиг на $k > n$ не имеет смысла, т. к. его результат такой же, как и при $k = n$).

Операция *"сдвиг вправо логический"* `SRL A` (от англ. *Shift Right Logical*) выполняется аналогично (рис. 2.2) и переводит исходную битовую строку в $0a_{n-1}a_{n-2}...a_1$. В машинах, где есть операции с параметром `SRL A,k`, есть и `SLL A,k`. Так как при сдвигах часть битов теряется, оказывается, что

`SRL(SLL A,k), k ≠ A`,

так же как `SLL(SRL A,k), k ≠ A`.

Рис. 2.2 Логический сдвиг вправо

В некоторых компьютерах имеется операция *"сдвиг влево циклический"* — `SLC`. Она выполняется, как показано на рис. 2.3, и переводит строку в $a_{n-2}a_{n-3}...a_0a_{n-1}$. Аналогично выполняется операция `SRC`.

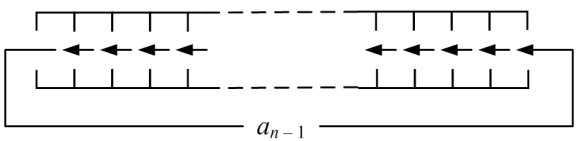


Рис. 2.3. Циклический сдвиг влево

В некоторых машинах имеется операция с параметром — `SLC A,k`. При этом операции `SRC A,k` может и не быть, т. к. `SLC A,k = SRC A,n-k`. Встречается операция *"обмен"*. Она меняет местами левую половину битовой строки с правой по схеме (рис. 2.4).

Эта операция эквивалентна `SLC A,n/2`.

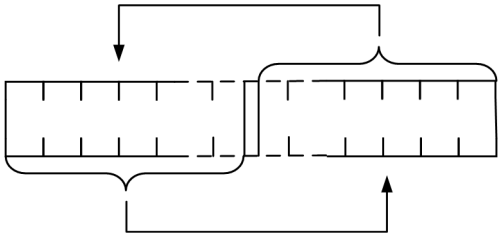


Рис. 2.4. Обмен

2.1.2. Операции математической логики

Эти операции называют также *булевыми операциями*, а переменные, с которыми эти операции производят, называют *булевыми переменными*. Булевы переменные отличаются от обычных тем, что могут принимать только два значения: 0 и 1. В булевой алгебре рассматриваются различные булевы функции от одной, двух и более переменных. Булева функция, во-первых, может быть вычислена только от булевых переменных, а во-вторых, сама может принимать только булевы значения — 0 или 1. Количество различных булевых функций от k переменных конечно.

Рассмотрим функции одной переменной: $y = f(x)$. Их всего 4 (табл. 2.1).

Таблица 2.1. Булевы функции одной переменной

Значения x	**Значения функций $y_i = f_i(x)$**			
	f_0	f_1	f_2	f_3
0	0	0	1	1
1	0	1	0	1

Первую из этих функций можно записать как $f_0(x)$, последнюю — как $f_3(x)$, вторую — как $f_1(x)$. Все они неинтересны.

Лишь третья используется в некоторых компьютерах. Ее называют инверсией или отрицанием, или просто "НЕ". Действительно, если $y = f_2(x)$, то это значит, что y противоположно x, отрицает его. $f_2(x)$ имеет специальное обозначение: $\overline{x}$ или $\neg x$. В программах обычно пишут NOT x.

Существует 16 различных булевых функций двух переменных (табл. 2.2).

Таблица 2.2. Булевы функции двух переменных

Значения пар (x, y)	Значения функций $z = f_i(x, y)$				
	f_0	f_1	f_2	...	f_{15}
0, 0	0	0	0	...	1
0, 1	0	0	0	...	1
1, 0	0	0	1	...	1
1, 1	0	1	0	...	1

Заметим, что значения f_i, прочитанные сверху вниз, как единое двоичное число дают i. Например, значения f_2 составляют число $0010 = 2_{10}$, значения f_{15} составляют число $1111_2 = 15_{10}$. Число различных комбинаций значений k аргументов равно 2^k, следовательно, число различных булевых функций k аргументов равно 2^k.

Практически во всех компьютерах имеются операции, реализующие три булевых функции двух переменных, с вариантами названий:

- "И", "логическое умножение", "конъюнкция";
- "ИЛИ", "логическое сложение", "дизъюнкция";
- "исключающее ИЛИ", "сложение по модулю 2".

Их обозначения:

- $z = x \wedge y$ или $z = x$ AND y;
- $z = x \vee y$ или $z = x$ OR y;
- $z = (x + y) \bmod 2$ или $z = x$ XOR y.

Для описания этих функций используют так называемые *таблицы истинности* (рис. 2.5).

Очевидно, что это соответствует функциям предыдущей таблицы f_1, f_7, f_6.

Можно пользоваться также словесными описаниями:

- только если x и y одновременно равны 1, $z = 1$;
- только если x и y одновременно равны 0, $z = 0$;
- только если $x = y$, $z = 0$.

Все остальные булевы функции двух аргументов легко получаются как комбинации этих трех.

x \ y	0	1
0	0	0
1	0	1

$z = x \wedge y$

x \ y	0	1
0	0	1
1	1	1

$z = x \vee y$

x \ y	0	1
0	0	1
1	1	0

$z = (x + y) \bmod 2$

Рис. 2.5. Таблицы истинности

Операции математической логики в компьютерах реализованы как групповые: аргументами операций являются битовые строки, и операция выполняется отдельно над каждой парой битов. Если

$$x = a_{n-1}a_{n-2}\ldots a_0\,,\ \ y = b_{n-1}b_{n-2}\ldots b_0\,,$$

то результат любой из бинарных операций математической логики $z = x * y$ — битовая строка вида

$$z = c_{n-1}c_{n-2}\ldots c_0\,,\ \text{где}\ \ c_i = a_i * b_i\ \ \text{для}\ \ i = 0,\ \ 1,\ldots,\ \ n-1\,.$$

Здесь символ $*$ означает произвольную операцию математической логики. Вообще-то следовало бы говорить: "команда группового логического умножения", однако других команд логического умножения в компьютерах не бывает, поэтому говорят просто, хотя и неточно: "команда логического умножения". Аналогично обстоит дело и с другими командами математической логики.

Команды математической логики используются для обработки части ячейки — для выделения, стирания или изменения некоторых ее битов. Пусть, например, из информации, хранящейся в ячейке — $a_{n-1}a_{n-2}\ldots a_5a_4a_3a_2a_1a_0$, нас интересует в данный момент только ее часть, расположенная в битах 1, 2 и 4. Достаточно выполнить команду логического умножения этой битовой строки на строку вида

$$00...010110$$

(единицы в битах 1, 2 и 4, нули — в остальных), и мы получим новую строку

$$00...0a_4\,0a_2a_1\,0\,.$$

Пусть ячейка имеет размер байта и пусть ее содержимое — 101101101. Тогда в результате логического умножения этого байта на 000010110 получим 000000100. Пусть информация, хранящаяся в битах 1, 4, 5 и 7, устарела и должна быть заменена новой.

Замену можно выполнить в два приема:

1. Стирание устаревшей информации.
2. Запись новой информации.

Стирание выполним с помощью логического умножения. Для этого байт со старой информацией — $a_7a_6a_5a_4a_3a_2a_1a_0$ умножим на байт вида 01001101 (нули в позициях 1, 4, 5, 7, в остальных — единицы). Результат — $0a_600a_3a_20a_0$. Запись новой информации выполняем логическим сложением полученного байта с битовой строкой $b_70b_5b_400b_10$ (b_7, b_5, b_4, b_1 — новая информация). Результат — $b_7a_6b_5b_4a_3a_2b_1a_0$.

Команду "исключающее ИЛИ" можно использовать, например, для инверсии части битов ячейки. Результатом применения этой операции к байтам $a_7a_6a_5a_4a_3a_2a_1a_0$ и 01100100 будет $a_7\overline{a}_6\overline{a}_5a_4a_3\overline{a}_2a_1a_0$, т. е. инверсия байтов 2, 5 и 6.

2.1.3. Маски

Давно известен простой способ шифрования информации с помощью шаблона. Пусть, к примеру, надо записать короткое сообщение — имя из пяти букв — в строку, имеющую двенадцать позиций. Если имя длиннее пяти букв, будем лишние отбрасывать, если короче, — дополнять точками. Используем шаблон — полоску бумаги, на которой нарисованы 12 клеточек (рис. 2.6).

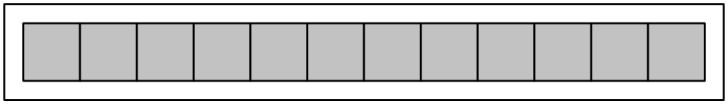


Рис. 2.6

Вырежем отверстия на месте пяти клеточек, выбранных произвольно. Например так, как показано на рис. 2.7.

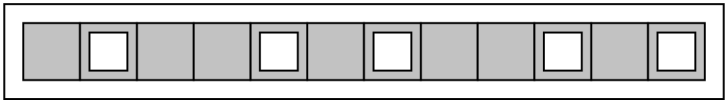


Рис. 2.7. Шаблон (маска)

Теперь наложим наш шаблон на чистый лист бумаги, предназначенный для записи сообщения, и впишем это сообщение в отверстия (рис. 2.8).

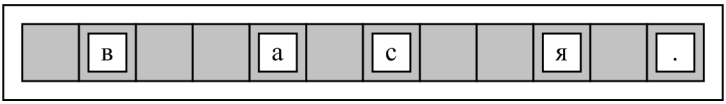


Рис. 2.8

Снимем шаблон и в пустые позиции впишем любые буквы и знаки (рис. 2.9).

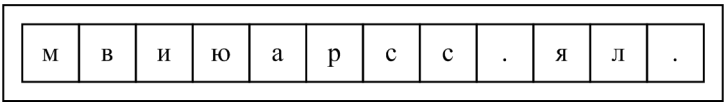


Рис. 2.9. Конечный вид

Прочитать это сообщение может лишь тот, кто имеет шаблон. Наложив шаблон, мы закроем все ненужные буквы.

Вообще-то совсем не обязательно прятать сообщение среди "мусора". Можно в эту же строку записать сообщение из 12 букв, используя, например, последовательно три шаблона по четыре буквы. Сообщение может быть записано и прочитано с помощью трех шаблонов.

Шаблоны могут использоваться и для того, чтобы скрыть часть информации.

Например, подробные анкетные данные сотрудников, хранящиеся в отделе кадров, являются частично конфиденциальной информацией. Домашний адрес и некоторые другие данные не сообщаются посторонним, в то время как рабочий телефон, номер кабинета и т. д. — открытая информация. Можно изготовить шаблон, наложив который на анкету мы закроем, замаскируем конфиденциальную информацию и оставим доступной открытую.

В связи с такой функцией шаблона его называют также *маской*.

В компьютерах в качестве маски может быть использована любая строка битов. При этом единицы в строке играют роль прорезей в шаблоне. Так, например, если прочитать

байт	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
по маске	0	0	1	0	1	0	0	0
то получим								
	0	0	a_5	0	a_3	0	0	0

Заметим, что эта операция эквивалентна логическому умножению байта на маску. Ее называют "прочитать по маске".

Используется также операция "собрать по маске". Она "читает по маске" и сдвигает прочитанное в левый край байта-результата. Остальные биты за-

полняются нулями. Например: байт 11011000 собрать по маске 00101000 (рис. 2.10).

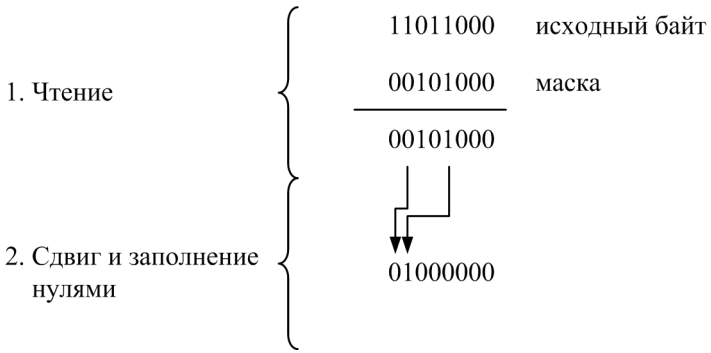


Рис. 2.10. Операция "собрать по маске"

Команда "разобрать по маске" последовательно берет биты исходного байта, начиная с левого, и расставляет их в позиции, помеченные в маске единицами. Остальные позиции заполняются нулями (рис. 2.11).

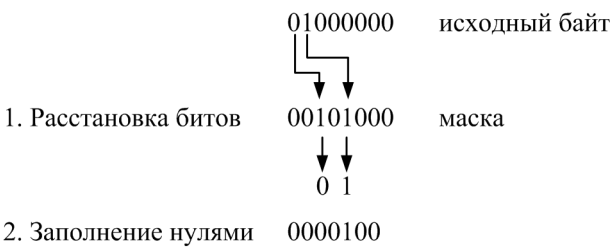


Рис. 2.11. Операция "разобрать по маске"

2.2. Арифметика целых

Двоичная система хороша не только тем, что записанные в ней числа удобно хранить, но и тем, что выполнение арифметических операций в двоичной системе чрезвычайно просто.

Для сложения многозначных двоичных чисел достаточно использовать таблицу сложения (табл. 2.3) и обычное правило переноса единицы в старший разряд. (Аналогичная таблица для десятичной системы имеет размер 10×10.)

Для умножения "в столбик" таблицы умножения и вовсе не требуется (рис. 2.12).

Таблица 2.3. z = x + y

x \ *y*	**0**	**1**
0	0	1
1	1	10

```
      1011
  ×   1001
  ------
      1011
+  1011
  ------
  1100011
```

Рис. 2.12. Двоичное умножение

Как видно из примера, если в разряде множителя 1, то множимое просто переписывается под черту с соответствующим сдвигом, если 0, то пропускается.

Особенностью компьютерной арифметики является проблема переполнения. Она заключается в том, что для записи результата любой арифметической операции заранее отводится ячейка определенного размера, с определенным количеством разрядов, но результат может содержать больше разрядов, а значит, может не помещаться в ячейку. Обычно это означает, что дальнейшие вычисления не имеют смысла. В этом случае выполнение программы надо закончить и выдать сообщение о переполнении. В некоторых случаях программист знает, что может произойти переполнение, но считает возможным продолжить вычисления при условии, что факт переполнения станет ему известен. Так или иначе, но оказывается необходимым, и в компьютере вырабатывается, специальный признак переполнения, если результат не помещается в отведенное для него место.

Иногда можно заранее, до выполнения вычислений, сказать, что переполнения не будет. (Например, при вычитании положительных чисел.) Поэтому, описывая арифметические операции, обычно приводят условия возникновения переполнения.

2.2.1. Операции с беззнаковыми числами

Иногда числа, хранящиеся в ячейках памяти, интерпретируются как целые без знака. Пару таких чисел можно сложить или вычесть из одного другое.

При этом результат также интерпретируется как беззнаковый. (В некоторых машинах такие операции реализованы.)

При сложении может возникнуть *переполнение*. Оно возникает всякий раз, когда сумма больше $2^n - 1$. При вычитании переполнение получиться не может, однако, если вычитаемое больше, чем уменьшаемое, возникает похожая ситуация: результат операции должен быть беззнаковым, а получить таковой невозможно.

Еще один тип операций с беззнаковыми числами — сложение n-значных чисел по модулю 2^m или по модулю $2^m - 1$. Общее обозначение операций сложения по модулю A:

$$z = (x + y) \bmod A.$$

Алгоритм сложения по модулю A:

1. Вычисляется сумма $x + y$.
2. Сумма делится на A.
3. Результатом объявляется остаток от деления.

Результат операций этого типа всегда неотрицателен. Переполнение возникнуть не может.

Сложение по модулю 2^m — частный случай описанных операций и выполняется проще:

1. Вычисляется сумма $x + y$.
2. В качестве результата берутся m младших цифр суммы.

Действительно, пусть сумма двух n-значных чисел имеет вид

$$c_{n-1}c_{n-2}\ldots c_m c_{m-1}\ldots c_0.$$

Поделив ее на 2^m, получим (рис. 2.13).

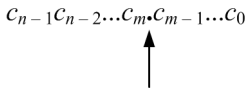


Рис. 2.13

Указанная стрелкой точка разделяет целую и дробную части суммы, т. е. получилось

$$c_{n-1}c_{n-2}...c_m + \frac{c_{m-1}...c_0}{2^m}.$$

Следовательно, $c_{m-1}...c_0$ — остаток. В компьютерах обычно $m = n$. В этом случае для вычисления остатка необходимо лишь отбросить c_n, если он образовался. То есть такую операцию можно интерпретировать как сложение с игнорированием переполнения.

Сложение m-значных чисел по модулю $2^m - 1$ выполняется так:

1. Вычисляется сумма $z_1 = x + y$.
2. Если $z_1 < 2^m$, то $z = z_1$.
3. Если $z_1 \geq 2^m$, т. е. z_1 состоит из $m+1$ бита, то отбрасывается левый бит и к результату прибавляется 1.

Иными словами

$$z = \begin{cases} x + y, & \text{если } x + y < 2^m; \\ x + y - 2^m + 1, & \text{если } x + y \geq 2^m. \end{cases}$$

Этот вид сложения называют *циклическим*, его реализуют с помощью циклического счетчика (рис. 2.14).

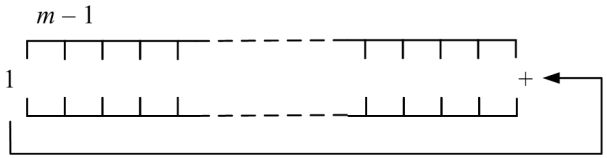


Рис. 2.14. Циклическое сложение

В обычном счетчике, если $x + y \geq 2^m$, старшая единица результата теряется, — ее некуда передать. В циклическом счетчике единица переноса из старшего разряда, если она образовалась, прибавляется к младшему разряду.

2.2.2. Сложение и вычитание целых в прямом коде. Сравнение и признаки результата

Для выполнения сложения целых чисел в прямом коде необходимо сначала проверить, имеют ли они одинаковые знаки. Если да, то абсолютные величины этих чисел складываются и сумме присваивается их знак. В противном случае абсолютные величины нужно вычесть (из большей меньшую) и разности присвоить знак большей.

Пусть x и y — слагаемые, а z — их сумма. Введем обозначения для x:

$$x = \sigma_x \times |x|, \quad \sigma_x = \begin{cases} 1 & \text{при } x \geq 0; \\ -1 & \text{при x} < 0. \end{cases}$$

И аналогичные для y. Теперь опишем этот алгоритм более формально.

Алгоритм ADD (от англ. *add* — сложить):

1. Если $\sigma_x \times \sigma_y = 1$, то перейти к пункту 2, иначе — к пункту 3.
2. $z := \sigma_x \times (|x| + |y|)$. Конец.
3. Если $|x| > |y|$, то перейти к пункту 4, иначе — к пункту 5.
4. $z := \sigma_x \times (|x| - |y|)$. Конец.
5. $z := \sigma_y \times (|y| - |x|)$. Конец.

Здесь употреблен знак := — присвоить.

Вычитание знаковых чисел заменяется сложением после изменения знака вычитаемого на противоположный.

Алгоритм SUB (от англ. *subtract* — вычесть):

1. $\sigma_y := -\sigma_y$.
2. $z := x + y$ (выполнить алгоритм ADD).

Заметим, что для реализации этого алгоритма надо уметь определять, какое из двух чисел по абсолютной величине больше. Мы делаем это очень легко, "с одного взгляда", однако для компьютера это не так-то просто. Надо выполнять специальный алгоритм, заключающийся в следующем.

Сравнивают крайние левые разряды чисел x и y (не знаковые); если они совпадают, то переходят на одну позицию вправо и сравнивают следующую пару разрядов. Если и они совпадают, то переходят еще на один разряд вправо, и т. д. Обнаружив несовпадающую пару разрядов, объявляют бóльшим то число, в котором соответствующий разряд равен 1.

Сравнение является частью некоторых арифметических операций, и в то же время обычно имеется и самостоятельная операция "сравнение". Ее задача — определить, какое из двух чисел, x или y, больше. Результат операции можно представить в виде двухбитового числа *CC* (от англ. *condition code*; в отечественной литературе его называют *признаком результата*), определяемого по правилу

$$CC = \begin{cases} 0 & \text{при } x = y; \\ 1 & \text{при } x > y. \end{cases}$$

Можно сказать, что операция выполняется как вычитание. При этом, однако, сам результат вычитания не фиксируется, а сохраняется только его признак.

Алгоритм CMP (от англ. *compare* — сравнивать):

1. $CC := \sigma_{x-y}$. Конец.

Признак результата нужен для управления разветвлениями алгоритма. С другой стороны, он так легко получается как побочный результат арифметических операций, что в большинстве компьютеров эти операции устроены таким образом, что наряду с самим результатом отдельно фиксируют также его признак. Далее приведены соответствующие алгоритмы сложения и вычитания.

Алгоритм ADD (вариант 2):

1. Если $\sigma_x \times \sigma_y = 1$, то перейти к пункту 2, иначе — к пункту 3.
2. $z := \sigma_x \times (|x| + |y|)$. Перейти к пункту 6.
3. Если $|x| > |y|$, то перейти к пункту 4, иначе — к пункту 5.
4. $z := \sigma_x \times (|x| - |y|)$. Перейти к пункту 6.
5. $z := \sigma_y \times (|y| - |x|)$.
6. $CC := \sigma_z$. Конец.

Алгоритм SUB (вариант 2):

1. $\sigma := -\sigma_y$.
2. $z := x + y$.
3. $CC := \sigma_z$. Конец.

Кстати, при переполнении, которое может возникать как результат арифметических операций, фиксируется признак результата $CC = 3$.

2.2.3. Сложение и вычитание в дополнительном коде

В операциях сложения и вычитания дополнительный код числа рассматривается как единое n-битное беззнаковое число, т. е. определяется формулой

$$x_{\text{доп}} = \begin{cases} x_{\text{пр}} & \text{при } x \geq 0; \\ 2^n - |x_{\text{пр}}| & \text{при } x < 0. \end{cases}$$

Операции выполняются над кодами в целом, т. е. складываются и вычитаются беззнаковые n-битные коды.

Будем обозначать через z число, равное сумме чисел x и y: $z = x + y$, через z^* — код, который получится при сложении дополнительных кодов x и y: $z^* = x_{\text{доп}} + y_{\text{доп}}$. Обозначим биты дополнительного кода числа x через a_i, числа y — через b_i, числа z — через c_i:

$$x_{\text{доп}} = a_{n-1}a_{n-2}\ldots a_0,\ y_{\text{доп}} = b_{n-1}b_{n-2}\ldots b_0,\ z_{\text{доп}} = c_{n-1}c_{n-2}\ldots c_0.$$

Естественно, результат может иметь и $n+1$ битов, но это не всегда означает переполнение. Например, при сложении положительного числа с отрицательным переполнение исключено, однако, поскольку $b_{n-1} = 1$, возможно, что и $c_n = 1$; при сложении же двух отрицательных чисел всегда $c_n = 1$, т. к. a_{n-1} и b_{n-1} равны 1. Поэтому будем отдельно рассматривать три случая:

1. $x \ge 0,\ y \ge 0$.
2. $x \ge 0,\ y < 0$.
3. $x < 0,\ y < 0$.

Случай $x < 0,\ y \ge 0$ аналогичен случаю 2.

При этом будем решать две проблемы:

- как по $(n+1)$-битному коду z^* получить $z_{\text{доп}}$;
- как определить, что наступило переполнение.

Для решения второй проблемы будем следить за состоянием битов c_n и c_{n-1}.

Случай 1 ($x \ge 0,\ y \ge 0$). Так как оба слагаемых положительны, результат тоже положительный, причем может оказаться и больше максимально представимого — $2^{n-1} - 1$. Так как для положительных чисел прямой и дополнительный коды совпадают, то $z^* = x_{\text{пр}} + y_{\text{пр}}$. Здесь возможны два варианта.

- Если $z^* < 2^{n-1}$, то $c_{n-1} = 0$ (а c_n — тем более), следовательно, z^* удовлетворяет формальным требованиям прямого кода для положительных чисел. Поэтому, $z^* = z_{\text{пр}}$. А так как дополнительный код положительных чисел совпадает с их прямым кодом, то $z^* = z_{\text{доп}}$.
- Если, $z^* \ge 2^{n-1}$, то его прямой код не может быть размещен в ячейке из n битов, следовательно, должно быть зафиксировано переполнение. Так как x и y меньше 2^{n-1}, $z^* < 2^n$. Поэтому $c_n = 0$, а $c_{n-1} = 1$.

Случай 2 ($x \ge 0,\ y < 0$). Сумма может быть как положительной, так и отрицательной. Результат всегда представим в ячейке, т. е. лежит в диапазоне $-2^{n-1} \le z \le 2^{n-1} - 1$. При сложении кодов получим

$$z^* = x_{\text{доп}} + y_{\text{доп}} = x_{\text{пр}} + 2^n - \left|y_{\text{пр}}\right| = 2^n + \left(x_{\text{пр}} - \left|y_{\text{пр}}\right|\right) = 2^n + z_{\text{пр}}.$$

- Если $z_{\text{пр}} \ge 0$, то $z_{\text{пр}} = z_{\text{доп}}$. В то же время $z^* \ge 2^n$. Следовательно, $c_n = 1$. Но $z_{\text{доп}}$ может быть получено из z^* отбрасыванием c_n. Поскольку $z_{\text{пр}} \le 2^{n-1} - 1,$ то $c_{n-1} = 0.$
- Если $z_{\text{пр}} < 0$, то $z^* = 2^n - \left|z_{\text{пр}}\right|$, что по определению является дополнительным кодом z. То есть $z^* = z_{\text{доп}}$. Так как $z_{\text{пр}} \ge 2^{n-1},$ то оказывается, что $c_n = 0,$ а $c_{n-1} = 1.$

Случай 3 $(x < 0,\ y < 0)$.

$$z^* = x_{\text{доп}} + y_{\text{доп}} = 2^n - \left|x_{\text{пр}}\right| + 2^n - \left|y_{\text{пр}}\right| = 2^n + [\left[2^n - \left(\left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right|\right)\right].$$

- Если $\left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right| < 2^{n-1}$, то прямой n-битный код суммы существует и $\left|z_{\text{пр}}\right| = \left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right|,$ а $z^* = 2^n + \left[2^n - \left(\left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right|\right)\right].$ Следовательно, $z^* = 2^n + z_{\text{доп}},$ $c_n = 1,$ $c_{n-1} = 1$ и $z_{\text{доп}}$ может быть получено из z^* отбрасыванием c_n.
- Если $\left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right| \ge 2^{n-1}$, должно фиксироваться переполнение. При этом, т. к. $\left|x_{\text{пр}}\right| + \left|y_{\text{пр}}\right| < 2^n$, оказывается, что $c_n = 1,$ $c_{n-1} = 0$.

Подводя итоги полученным результатам, констатируем, что во всех случаях реализуется одна из трех возможностей:

1. Сумма дополнительных кодов является дополнительным кодом суммы, т. е. $z_{\text{доп}} = z^*$.
2. Дополнительный код суммы получается из суммы дополнительных кодов отбрасыванием c_n, т. е. $z_{\text{доп}} = z^* - 2^n$.
3. Дополнительный код суммы не может быть получен, т. к. сумма выходит за диапазон представимых чисел, должно быть зафиксировано переполнение.

Переполнение легко обнаружить по противоречию: если сумма положительных чисел имеет признак отрицательного числа ($c_{n-1}=1$) или сумма отрицательных чисел — признак положительного ($c_{n-1}=0$), то имеет место переполнение. В табл. 2.4 показано, как нужно интерпретировать результат в зависимости от значения его двух крайних левых битов c_n и c_{n-1} и знаковых битов слагаемых a_{n-1} и b_{n-1}.

Таблица 2.4. Интерпретация результата

c_n, c_{n-1} / a_{n-1}, b_{n-1}	**0, 0**	**0, 1**	**1, 0**	**1, 1**
0, 0	$z_{\text{доп}} = z*$	Переполнение	Не бывает	Не бывает
0, 1	Не бывает	$z_{\text{доп}} = z^*$	$z_{\text{доп}} = z^* - 2^n$	Не бывает
1, 1	Не бывает	Не бывает	Переполнение	$z_{\text{доп}} = z^* - 2^n$

Таким образом, общее правило вычисления суммы в дополнительном коде:

1. Вычислить сумму дополнительных кодов слагаемых.
2. Если знаки слагаемых одинаковы, а левые биты результата (c_n и c_{n-1}) различны, то фиксировать переполнение.
3. В противном случае в качестве результата взять n младших битов суммы.

Очевидно, что этот алгоритм проще, чем алгоритм сложения в прямом коде.

Иногда признак переполнения формулируют иначе.

Сложение беззнаковых чисел фактически выполняется в два этапа: на первом одноименные разряды складываются с формированием единиц переноса, на втором прибавляются единицы переноса. Так вот, число переносов подсчитывается. Правда, не всех, а только число переносов в $(n-1)$-й и в n-й разряды. Очевидно, что это число может быть 0, 1 или 2. По нашей таблице нетрудно установить, что при нечетном числе переносов переполнения нет, а при четном — есть.

Рассмотрим примеры сложения чисел, расположенных в коротких ячейках — из 8 битов (обычно целые со знаком записываются в ячейки из 16 битов или более, но для примеров удобнее короткие ячейки).

СЛУЧАЙ 1.

Пример 1 (рис. 2.15).

$$
\begin{array}{lll}
x = 111011 & y = 10011 & \ 00111011 \\
x_{\text{пр}} = 00111011 & y_{\text{пр}} = 00010011 & +\ \ 00010011 \\
x_{\text{доп}} = 00111011 & y_{\text{доп}} = 00010011 & z^* = 01001110 \\
\uparrow & \uparrow & \uparrow \\
a_{n-1} = 0 & b_{n-1} = 0 & c_{n-1} = 0
\end{array}
$$

Рис. 2.15

Переполнения нет, $z_{\text{доп}} = z^* = 0100$.

Пример 2 (рис. 2.16).

$$
\begin{array}{lll}
x = 111011 & y = 1010011 & \ 00111011 \\
x_{\text{пр}} = 00111011 & y_{\text{пр}} = 01010011 & +\ \ 01010011 \\
x_{\text{доп}} = 00111011 & y_{\text{доп}} = 00010011 & z^* = 10001110 \\
\uparrow & \uparrow & \uparrow \\
a_{n-1} = 0 & b_{n-1} = 0 & c_{n-1} = 1
\end{array}
$$

Рис. 2.16

Переполнение.

Пример 3 (рис. 2.17).

$$
\begin{array}{lll}
x = 1111111 & y = 1111111 & \ 01111111 \\
x_{\text{пр}} = 01111111 & y_{\text{пр}} = 01111111 & +\ \ 01111111 \\
x_{\text{доп}} = 01111111 & y_{\text{доп}} = 01111111 & z^* = 11111110 \\
\uparrow & \uparrow & \uparrow \\
a_{n-1} = 0 & b_{n-1} = 0 & c_{n-1} = 1
\end{array}
$$

Рис. 2.17

Переполнение. Даже при самых больших x и y $c_n = 0$.

СЛУЧАЙ 2.

Пример 4 (рис. 2.18).

$$\begin{array}{llll} x = -111011 & y = 10011 & & 11000101 \\ & & & + \\ x_{\text{пр}} = 10111011 & y_{\text{пр}} = 00010011 & & 00010011 \\ x_{\text{доп}} = 11000101 & y_{\text{доп}} = 00010011 & & z^* = 11011000 \\ \uparrow & \uparrow & & \uparrow \\ a_{n-1} = 1 & b_{n-1} = 0 & & c_{n-1} = 1 \end{array}$$

Рис. 2.18

Переполнения нет, $z_{\text{доп}} = z^* = 1101$.

Пример 5 (рис. 2.19).

$$\begin{array}{llll} x = 111011 & y = -10011 & & 00111011 \\ & & & + \\ x_{\text{пр}} = 00111011 & y_{\text{пр}} = 10010011 & & 11101101 \\ x_{\text{доп}} = 00111011 & y_{\text{доп}} = 11101101 & & z^* = 100101000 \\ \uparrow & \uparrow & & \uparrow \\ a_{n-1} = 0 & b_{n-1} = 1 & & c_{n-1} = 0 \end{array}$$

Рис. 2.19

Переполнения нет, $z_{\text{доп}} = z* - 2^8 = 00101000$, $z_{\text{пр}} = z_{\text{доп}}$, $z = 101000$. Конечно, вычитания 2^8 в машине не производится, — просто отбрасывается c_n.

СЛУЧАЙ 3.

Пример 6 (рис. 2.20).

Переполнения нет, $z_{\text{доп}} = z^* - 2^8 = 10110010$, $z_{\text{пр}} = 11001110$, $z = -1001110$.

Пример 7 (рис. 2.21).

Переполнение.

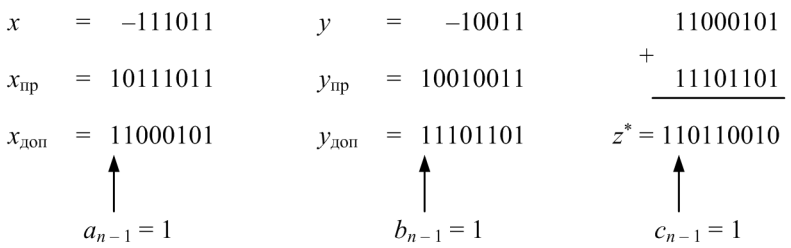


Рис. 2.20

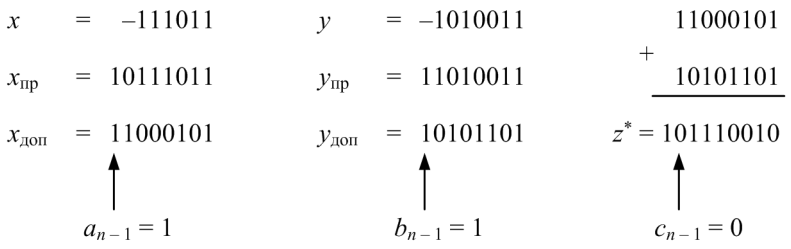


Рис. 2.21

2.2.4. Умножение и деление целых чисел

При умножении может возникнуть переполнение. Причем происходит это значительно чаще, чем при сложении или вычитании. Обычно в компьютерах умножение целых устроено таким образом, что результат умножения содержимого двух ячеек записывается в ячейку двойной длины. Максимальное число в ячейке из n битов — $2^{n-1}-1$, следовательно, произведение не превышает $\left(2^{n-1}-1\right)^2 < 2^{2n-2}-1$, а значит, число разрядов произведения не больше $2n-2$. В ячейке двойной длины для записи числа отводится $2n-1$ битов. В таком случае переполнение возникнуть не может. Число выравнивается при записи по правому краю, т. е. записывается в младшие разряды сдвоенной ячейки. Так как числа записываются в дополнительном коде, бит, следующий за знаком, заполняется по-разному для положительных и отрицательных чисел. У положительных чисел он равен нулю, у отрицательных — единице.

Существует несколько вариантов алгоритмов умножения целых чисел со знаком.

Самый простой из них:

1. Если хотя бы одно из чисел положительно — перемножаются их коды. Умножение выполняется по той же схеме "в столбик", что и в десятичной

системе — вычисляем частичные произведения умножением множимого последовательно на цифры множителя и суммируем частичные произведения с соответствующим сдвигом. Разница только в том, что фактически умножения нет, — есть простое многократное копирование множимого со сдвигом.

2. Если оба числа отрицательны — они переводятся в прямой код и затем перемножаются.

На практике обычно применяются более быстрые алгоритмы умножения, являющиеся модификациями алгоритма Бута. В действительности этот алгоритм известен очень давно как прием устного счета. Если требуется в уме помножить какое-то число на 990, то проще помножить его на 1000 и затем вычесть множимое, умноженное на 10. В десятичной системе этот прием используется редко, — когда встретится сочетание нескольких девяток подряд. В двоичной системе сочетание нескольких единиц подряд в множителе — обычное явление.

Предположим, множитель имеет вид $x = 0011100011100000$, т. е. $x = 2^{13} + 2^{12} + 2^{11} + 2^7 + 2^6 + 2^5$. Умножение на такое число потребует сложения шести слагаемых — частичных сумм. В алгоритме Бута множитель заменяют числом $x = 2^{14} - 2^{11} + 2^8 - 2^5$. Используя обозначение $\overline{1} = -1$, можно записать $x = 000100\overline{1}00100\overline{1}00000$. При умножении на такое число следует частичные произведения на $\overline{1}$ брать в дополнительном коде.

Для деления чисел в дополнительном коде нет удобного алгоритма. Поэтому операция деления выполняется в три приема.

1. Операнды переводятся в прямой код.
2. Выполняется деление модулей чисел.
3. Результат записывается в дополнительном коде.

При делении целых получаются два результата — частное и остаток. Причем остаток имеет знак делимого. Например,

1. $7/2 = 3$, ост. $= 1$; $(-7)/2 = -3$, ост. $= -1$.
2. $7/(-2) = -3$, ост. $= 1$; $(-7)/(-2) = 3$, ост. $= -1$.

Это правило следует из определения: "поделить с остатком" x на y означает найти такие z_1 и z_2, что $x = y \times z_1 + z_2$.

Обычно деление целых в компьютерах строится как операция, обратная умножению, т. е. делимым является содержимое двойной ячейки, а делителем — простой. Частное и остаток также помещаются в простые ячейки. Поэтому возможно переполнение. При попытке деления на ноль также фиксируется переполнение.

2.2.5. Арифметический сдвиг

Арифметический сдвиг похож на логический. Отличие состоит в том, что знаковый бит здесь рассматривается самостоятельно, т. е. его участие в операции иное, чем других битов.

При сдвиге влево (обозначение `SLA A`) знаковый бит остается на месте, а остальная часть ячейки сдвигается так же, как при логическом сдвиге (рис. 2.22).

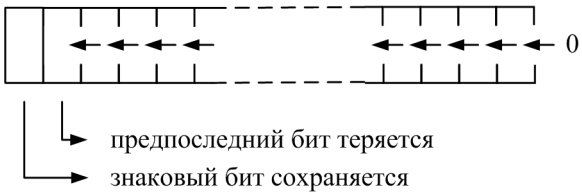


Рис. 2.22. Арифметический сдвиг влево

То есть, если $x = a_{n-1}a_{n-2}a_{n-3}...a_0$, то после сдвига получим $x = a_{n-1}a_{n-3}...a_0 0$.

Если $|x| < 2^n - 2$, то арифметический сдвиг влево эквивалентен умножению на 2.

Например, число 7 в ячейке из восьми битов имеет вид 00000111. После арифметического сдвига влево получим 00001110, что соответствует десятичному 14. Число –7 имеет вид 11111001 в дополнительном коде. После сдвига получим 11110010, что соответствует десятичному –14.

Если же $x \geq 2^{n-2}$, то происходит потеря старшей значащей цифры и результат не равен $2x$. Переполнение в этой операции не фиксируется.

В арифметическом сдвиге вправо (`SRA A`) участвуют все биты (рис. 2.23), но знаковый бит не меняется (при логическом сдвиге в него записывается 0).

Рис. 2.23. Арифметический сдвиг вправо

Таким образом, если $x = a_{n-1}a_{n-2}...a_0$, то после сдвига получим: $a_{n-1}a_{n-1}a_{n-2}...a_1$. Этот сдвиг эквивалентен делению на 2. Если число без остатка на 2 не делится ($a_0 = 1$), то получается $\frac{x-1}{2}$. Например, код числа 7 — 00000111, сдвинутый вправо дает 00000011 — число 3. Код числа –7 — 11111001 дает 11111100 — число –4.

В некоторых машинах имеются операции арифметического сдвига с параметром: `SLA A,k` и `SRA A,k` — сразу на k битов.

Умножение на различные степени двойки (положительные или отрицательные) выполняется во всех компьютерах намного быстрее с помощью арифметического сдвига, чем обычным путем.

2.3. Арифметика с плавающей точкой

В данном разделе рассматриваются операции вида $z = x * y$ (знак * — это обобщенное обозначение элементарных арифметических операций и заменяет знаки +, –, ×, /), а также $z = \sqrt{x}$, где x, y и z — числа с плавающей точкой, представленные в ячейках в виде троек σ_x, Π_x, m_x; σ_y, Π_y, m_y; σ_z, Π_z, m_z. При этом в ячейке из n разрядов один отведен под код знака числа — σ, он же — код знака мантиссы (мантисса записывается в дополнительном коде), k разрядов — под машинный порядок Π, r разрядов — под машинный код мантиссы m. Истинный порядок получается из машинного вычитанием смещения $p = \Pi - 2^{k-1}$, а истинная мантисса — добавлением точки и цифры, стоящей перед точкой. Обычно это 0, но иногда — 1:

$$M = 0.m \text{ или } M = 1.m .$$

Мы будем рассматривать три варианта интерпретации тройки σ_x, Π_x, m_x:

1. $x = 2^{p_x} \times \sigma_x \times 0.m_x$.
2. $x = 16^{p_x} \times \sigma_x \times 0.m_x$.
3. $x = 2^{p_x} \times \sigma_x \times 1.m_x$.

Биты мантисс m_x, m_y и m_z будем обозначать через a_i, b_i и c_i соответственно.

2.3.1. Сложение и вычитание

Эти операции выполняются в три этапа:

1. Выравнивание порядков.
2. Сложение (вычитание) мантисс.
3. Нормализация результата.

Пусть слагаемые имеют вид $x = 2^{p_x} \times \sigma_x \times 0.m_x$ и $y = 2^{p_y} \times \sigma_y \times 0.m_y$.

Если окажется, что $p_x = p_y$, то первый этап пропускается. В этом случае сумму можно представить в виде $z = 2^{p_x} \times \left(\sigma_x \times 0.m_x + \sigma_y \times 0.m_y\right)$.

Выражение, стоящее в скобках, вычисляется как сумма чисел с фиксированной точкой. В результате получим $z = 2^{p_x} \times \left|M_z\right| \times \sigma_z$. Это и есть второй этап. Так как слагаемые по модулю меньше 1 и могут иметь разные знаки, возможны три случая:

- $1 \le \left|M_z\right| < 2$;
- $\frac{1}{2} \le \left|M_z\right| < 1$;
- $\left|M_z\right| < \frac{1}{2}$.

Для каждого из них третий этап выполняется по-своему.

Самый простой случай — мантисса получилась нормализованной, $\left|M_z\right|$ имеет вид $0.c_{-1}c_{-2}...c_{-r}$, а код мантиссы результата $m_z = c_{-1}c_{-2}...c_{-r}$. Третий этап не требуется, и $p_z = p_x$, следовательно, $\Pi_x = \Pi_z$.

В случае $1 \le \left|M_z\right| < 2$ мантисса имеет вид $1.c_{-1}c_{-2}...c_{-r}$.

Требуется нормализация вправо — полученная сумма мантисс должна быть сдвинута на один бит вправо. Одновременно порядок результата увеличивается на 1. То есть выполняется преобразование $\left|z\right| = 2^{p_x+1} \times \left|M_{z1}\right|$, где $\left|M_{z1}\right| = \frac{\left|M_z\right|}{2}$. Теперь $\left|M_{z1}\right| = 0.1c_{-1}c_{-2}...c_{-r}$, а $\Pi_z = \Pi_x + 1$. Окончательно m_z получается из M_{z1} отбрасыванием знаков "0" и ".", а также c_{-r}, не умещающегося в разрядную сетку: $m_z = 1c_{-1}c_{-2}...c_{-r+1}$.

В случае $|M_z| < \frac{1}{2}$ требуется нормализация влево, причем количество битов, на которые следует сдвинуть мантиссу, может быть различным. Если $\frac{1}{4} \le |M_z| < \frac{1}{2}$, то оно равно 1, если $\frac{1}{8} \le |M_z| < \frac{1}{4}$, то 2. В общем случае, при $2^{-i-1} \le |M_z| < 2^{-1}$ требуется сдвиг на i битов (очевидно, столько нулей после запятой подряд имеет число $|M_z|$): $|M_z| = 0.00\ldots0c_{-i-1}c_{-i-2}\ldots c_{-r}$.

Одновременно на i уменьшается порядок. В результате $|z| = 2^{p_x - i} \times |M_{z2}|$, где $|M_{z2}| = 0.c_{-i-1}c_{-i-2}\ldots c_{-r}$.

Теперь результат нормализован, следовательно, $|m_z| = 0.c_{-i-1}c_{-i-2}\ldots c_{-r}00\ldots0$, $\Pi_z = \Pi_x - i$.

В конце числа — i нулей.

Вернемся к первому этапу. Пусть $\Pi_x > \Pi_y$. Тогда на первом этапе выполняется денормализация числа y за счет сдвига мантиссы вправо и соответствующего увеличения порядка. Величина сдвига равна разности $\Pi_x - \Pi_y$. Получаем $|y| = 2^{p_x} \times |M_{y1}|$, где $|M_{y1}| = 0.00\ldots0b_{-h-1}b_{-h-2}\ldots b_{-r}$, $h = \Pi_x - \Pi_y$. Порядки слагаемых выравниваются, и теперь можно выполнять второй этап так, как он описан. Если $\Pi_x < \Pi_y$, то аналогично преобразуется x.

Кратко алгоритм сложения можно записать так.

Этап 1. Выравнивание порядков.

1. Если $\Pi_x = \Pi_y$, то перейти к этапу 2.
2. Если $\Pi_x > \Pi_y$, то $|M_y| := 0.00\ldots0b_{-h-1}b_{-h-2}\ldots b_{-r}$ $(h = \Pi_x - \Pi_y)$, $\Pi_y := \Pi_x$ и перейти к этапу 2.
3. Если $\Pi_x < \Pi_y$, то $|M_x| := 0.00\ldots0a_{-h-1}a_{-h-2}\ldots a_{-r}$ $(h = \Pi_y - \Pi_x)$, $\Pi_x := \Pi_y$.

Этап 2. Сложение мантисс.

$$M_z := M_x + M_y.$$

Этап 3. Нормализация.

1. Если $|M_z| = 0.1c_{-2}\ldots c_{-r}$, то $m_z := 1c_{-2}\ldots c_{-r}$, $\Pi_z := \Pi_x$. Конец.
2. Если $|M_z| = 1.c_{-1}c_{-2}\ldots c_{-r}$, то $m_z := 1c_{-1}\ldots c_{-r+1}$, $\Pi_z := \Pi_x + 1$. Конец.
3. Если $|M_z| = 0.00\ldots01c_{-i-2}\ldots c_{-r}$, то $m_z = 1c_{-i-2}\ldots c_{-r}00\ldots0$, $\Pi_z = \Pi_x - i$. Конец.

В случае, если в машине принято представление вида $x = 16^{p_x} \times \sigma_x \times 0.m_x$, нормализация и денормализация могут выполняться только на тетрады. Описанный выше алгоритм сложения можно перенести и на этот случай, считая в нем всюду нули, единицы, a_i, b_i и c_i шестнадцатеричными цифрами. Конечно, одновременно r надо заменить на $\frac{r}{4}$ или ближайшим к $\frac{r}{4}$ целым, большим, чем $\frac{r}{4}$ (число тетрад в 4 раза меньше, чем число битов).

Заметим, что некоторые машины имеют разновидности команд сложения и вычитания с блокировкой нормализации. Точнее, их следовало бы называть с блокировкой нормализации влево, т. к. нормализация вправо заблокирована быть не может.

Если используется третий вариант представления $x = 2^{p_x} \times \sigma_x \times 1.m_x$ и $y = 2^{p_y} \times \sigma_y \times 1.m_y$, то сложение выполняется по схожему алгоритму.

Этап 1. Выравнивание порядков.

1. Если $\Pi_x = \Pi_y$, то перейти к этапу 2.
2. Если $\Pi_x > \Pi_y$, то $|M_y| := 0.00...01 b_{-h-1} b_{-h-2} ... b_{-r}$ $(h = \Pi_x - \Pi_y)$, $\Pi_y := \Pi_x$ и перейти к этапу 2.
3. Если $\Pi_x < \Pi_y$, то $|M_x| := 0.00...01 a_{-h-1} a_{-h-2} ... a_{-r}$ $(h = \Pi_y - \Pi_x)$, $\Pi_x := \Pi_y$.

Этап 2. Сложение мантисс.

$$M_z := M_x + M_y.$$

Этап 3. Нормализация.

1. Если $|M_z| = 1.c_{-1}c_{-2}...c_{-r}$, то $m_z := c_{-1}c_{-2}...c_{-r}$, $\Pi_z := \Pi_x$. Конец.
2. Если $|M_z| = 1c_0.c_{-1}c_{-2}...c_{-r}$, то $m_z := c_0 c_{-1}...c_{-r+1}$, $\Pi_z := \Pi_x + 1$. Конец.
3. Если $|M_z| = 0.00...01c_{-i-2}...c_{-r}$, то $m_z = c_{-i-2}...c_{-r}00\ldots0$, $\Pi_z = \Pi_x - i - 1$. Конец.

В конце — $i-1$ нуль.

Так как при таком представлении числа всегда должны быть нормализованы, блокировка нормализации здесь невозможна.

Замечания

- Во всех вариантах алгоритма на третьем этапе может потребоваться нормализация вправо. При этом порядок результата увеличивается на единицу. Если

порядок хотя бы одного из слагаемых уже был равен максимальному для данного представления, то наступает переполнение. При этом правильный порядок записан быть не может.

- Во всех трех вариантах при $x=-y$ получается $M_z=0$. В этом случае формальный признак — ноль в первом бите после точки, требует нормализации влево. Однако нормализация выполнена быть не может и в ячейку результата записывается тройка σ_x, Π_x, m_x (σ_x может оказаться любым). В машинах с обязательной нормализацией этот случай называют *антипереполнением* или *потерей значимости*. Для них это не истинный ноль (истинный ноль — при $\Pi_z=0$).

При нормализации влево может возникнуть такая ситуация: $M_z=0.00...0c_{-i-2}...c_{-r}$, $\Pi_x<i$. $i+1$ ноль после запятой.

Так как Π_z не может быть отрицательным, полная нормализация результата невозможна. Выполняется лишь частичная нормализация, т. е. мантисса сдвигается влево на Π_x битов и в качестве результата принимается $m_z=00...0c_{-i-2}...c_{-r}00...0$, $\Pi_z=0$.

Здесь $i-\Pi_x$ нулей в начале числа и Π_x — после.

В машинах, не допускающих работу с ненормализованными числами, такое число считается нулем (т. к. $\Pi_z=0$). При этом ненормализованность мантиссы не считается антипереполнением, поскольку в любых последующих операциях эта мантисса не будет использоваться.

2.3.2. Умножение

В общем виде алгоритм умножения можно записать так:

1. Сложение порядков.
2. Перемножение мантисс.
3. Определение знака.
4. Нормализация результата, если необходимо.

Действительно, перемножая $x=S^{p_x}\times M_x$ и $y=S^{p_y}\times M_y$, получим $z=S^{p_x+p_y}\times(M_x\times M_y)$.

Если $\frac{1}{2}\le\left|M_y\right|<1$ (при $S=2$), то для $\left|M_x\times M_y\right|$ может потребоваться нормализация влево, но не более, чем на один бит, т. к. $\frac{1}{4}\le\left|M_x\times M_y\right|<1$.

Если $\frac{1}{16} \le |M_x| < 1$, $\frac{1}{16} \le |M_y| < 1$ (при $S = 16$), то для $|M_x \times M_y|$ может потребоваться нормализация влево, но не более, чем на одну тетраду, т. к. $\frac{1}{256} \le |M_x \times M_y| < 1$.

В случае $1 \le |M_x| < 2$, $1 \le |M_y| < 2$ ($S = 2$) может потребоваться нормализация вправо, но не более, чем на один бит, т. к. $1 \le |M_x \times M_y| < 4$.

Краткая запись алгоритма умножения:

1. $\Pi_z := \Pi_x + \Pi_y - 2^{k-1}$ (подразумевается, что порядок записывается в k бит).
2. $|M_z| := |M_x| \times |M_y|$.
3. $\sigma_z := (\sigma_x + \sigma_y) \bmod 2$.
4. Для машин:
   - с $|M| < 1$:
     - если $|M_z| = 0.1c_{-2}c_{-3}...c_{-q}$, то $m_z := 1c_{-2}c_{-3}...c_{-q}$, иначе (если $|M_z| = 0.0c_{-2}c_{-3}...c_{-q}$) — $m_z := c_{-2}c_{-3}...c_{-q}0$, $\Pi_z := \Pi_z - 1$;
     - если $S = 2$, то $q = r$ — числу битов в мантиссе, а c_i — двоичные цифры, если $S = 16$, то $q = \frac{r}{4}$, а c_i — шестнадцатеричные;
   - с $|M| < 2$: если $|M_z| = 1.c_{-1}c_{-2}...c_{-r}$, то $m_z := c_{-1}c_{-2}...c_{-r}$, иначе (если $|M_z| = 1c_0.c_{-1}...c_{-r}$) — $m_z := c_0c_{-1}...c_{-r+1}$, $\Pi_z := \Pi_z + 1$.

Замечания

- В первом пункте алгоритма 2^{k-1} вычитается, чтобы получить машинный порядок произведения.
- Переполнение возникает, если в итоге порядок больше предельно допустимого: $\Pi_z \ge 2^k$.
- Может оказаться, что $\Pi_z < 0$. Тогда записывается $\Pi_z = 0$ — машинный ноль.

2.3.3. Деление

Алгоритм деления аналогичен алгоритму умножения:

1. Вычитание порядков.
2. Деление мантисс.
3. Определение знака.
4. Нормализация результата, если необходимо.

Очевидно, $z = S^{p_x - p_y} \times \frac{M_x}{M_y}$. Если в машине требуется, чтобы было $\frac{1}{2} \le M < 1$, то получим $\frac{1}{2} < \left|\frac{M_x}{M_y}\right| < 2$, т. е. может потребоваться нормализация вправо на один бит (но не более).

Если в машине $\frac{1}{16} \le M < 1$, то получим $\frac{1}{16} < \left|\frac{M_x}{M_y}\right| < 16$, т. е. может потребоваться нормализация вправо на одну тетраду (но не более).

Если же в машине $1 \le M < 2$, то $\frac{1}{2} < \left|\frac{M_x}{M_y}\right| < 2$, т. е. может потребоваться нормализация влево на один бит (но не более).

Краткая запись алгоритма деления.

1. $\Pi_z := \Pi_x - \Pi_y + 2^{k-1}$; $\left|M_z\right| := \left|\frac{M_x}{M_y}\right|$.
2. $\sigma_z := (\sigma_x + \sigma_y) \bmod 2$.
3. Для машин:
   - с $\left|M\right| < 1$:
     - если $\left|M_z\right| = 0.1c_{-2}c_{-3}...c_{-q}$, то $m_z := 1c_{-2}c_{-3}...c_{-q}$, иначе (если $\left|M_z\right| = 1.c_{-1}c_{-2}c_{-3}...c_{-q}$) — $m_z := 1c_{-1}c_{-2}...c_{-q+1}$, $\Pi_z := \Pi_z + 1$;
     - если $S = 2$, то $q = r$ — числу битов в мантиссе, а c_i — двоичные цифры, если $S = 16$, то $q = \frac{r}{4}$, а c_i — шестнадцатеричные;
   - с $\left|M\right| < 2$: если $\left|M_z\right| = 1.c_{-1}c_{-2}...c_{-r}$, то $m_z := c_{-1}c_{-2}...c_{-r}$, иначе (если $\left|M_z\right| = 0.1c_{-2}...c_{-r}$) — $m_z := c_{-2}...c_{-r}0$, $\Pi_z := \Pi_z - 1$.

Замечания

- При $\Pi_z \geq 2^k$ возникает переполнение.
- При $\Pi_z < 0$ возникает антипереполнение. При этом принимается $\Pi_z = 0$.
- Если $\Pi_y = 0$ или $M_y = 0$, операция не производится и считается, что произошло переполнение.

2.3.4. Квадратный корень

В некоторых машинах операция извлечения квадратного корня включается в число элементарных, в других — выполняется специальной программой.

Пусть $x = S^{p_x} \times M_x$. Будем опираться на очевидный факт:

- если $\frac{1}{4} \leq M_x < 1$, то $\frac{1}{2} \leq \sqrt{M_x} < 1$;
- если $\frac{1}{256} \leq M_x < 1$, то $\frac{1}{16} \leq \sqrt{M_x} < 1$;
- если $1 \leq M_x < 4$, то $1 \leq \sqrt{M_x} < 2$.

Для p_x четных будем вычислять $y = \sqrt{x}$ так: $y = S^{\frac{p_x}{2}} \times \sqrt{M_x}$.

При этом если мантисса x была нормализована, то в любом случае $\sqrt{M_x}$ будет нормализован и следует принять $p_y = \frac{p_x}{2}$, $M_y = \sqrt{M_x}$.

Для p_x нечетных надо выполнить преобразование:

- при $M_x < 1$ ($S = 2$ или 16) — $x = S^{p_x+1} \times \left(\frac{M_x}{S}\right)$;
- при $M_x < 2$ ($S = 2$) — $x = S^{p_x-1} \times (2M_x)$.

Теперь можно принять $p_y = \frac{p_x + 1}{2}$, $M_y = \sqrt{\frac{M_x}{S}}$ в первом и $p_y = \frac{p_x - 1}{2}$, $M_y = \sqrt{2M_x}$ во втором случае.

Краткая запись алгоритма:

1. Если Π_x — четное, то $\Pi_y := \frac{\Pi_x}{2} + 2^{k-2}$ (вспомним, что $p = \Pi - 2^{k-1}$), $M_y := \sqrt{M_x}$.

2. Если Π_x — нечетное, то

- $\Pi_y := \frac{\Pi_x + 1}{2} + 2^{k-2}, \quad M_y := \sqrt{\frac{M_x}{S}}$ при $M < 1$;
- $\Pi_y := \frac{\Pi_x - 1}{2} + 2^{k-2}, \quad M_y := \sqrt{2M_x}$ при $M < 2$.

Замечания

- Переполнение при выполнении этой операции не возникает, т. к. $\sqrt{x} < x$ при $x < 1$.
- Разумеется, должно быть $M_x \geq 0$. В противном случае операция невыполнима и ситуация приравнивается к переполнению.

2.4. Десятичная арифметика

В большинстве машин имеются команды сложения и вычитания десятичных чисел, записанных в двоично-десятичном коде (BDC). Очевидно, что без них можно обойтись — числа можно перевести в двоичную систему, в ней выполнить операцию, а затем перевести результат в десятичную систему. Получается, что возможны два способа организации вычислений (рис. 2.24 и 2.25).

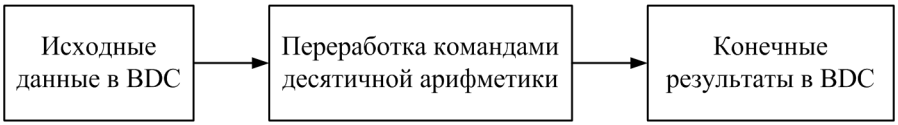


Рис. 2.24. Организация вычислений с помощью команд десятичной арифметики

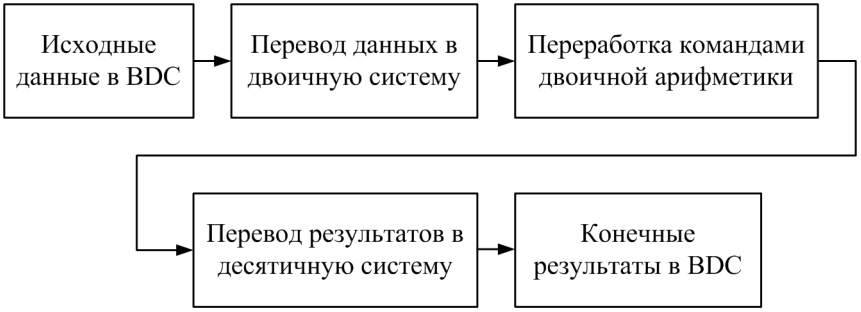


Рис. 2.25. Организация вычислений без команд десятичной арифметики

Хотя десятичные операции сложения и вычитания выполняются дольше, чем двоичные, перевод из десятичной системы в двоичную и обратно выполняется еще дольше. Следовательно, если числа вводятся в память в основном для хранения и лишь изредка складываются или вычитаются, то выгоднее операции выполнять в десятичной системе.

Это можно выразить соотношением

$$\nu \times t_{\text{десятичное}} > t_{10\to 2} + \nu \times t_{\text{двоичное}} + \mu \times t_{2\to 10},$$

где $t_{\text{двоичное}}$ и $t_{\text{десятичное}}$ — время выполнения процессором одной двоичной операции и одной десятичной, $t_{10\to 2}$ и $t_{2\to 10}$ — время, необходимое процессору для перевода одного числа из десятичной системы в двоичную и обратно, ν — среднее число арифметических операций, выполняемых процессором на каждое вводимое число, μ — среднее количество выводимых чисел, приходящееся на одно вводимое число. Если это соотношение выполнено, то лучше воспользоваться операциями десятичной арифметики.

Операции с десятичными числами основаны на следующих фактах:

- если шестнадцатеричное число x не превосходит 9, то оно совпадает с десятичным;
- если шестнадцатеричное x лежит в диапазоне $A_{16} \le x \le 13_{16}$, то десятичное x можно получить с помощью процедуры $x_{10} := x_{16} + 6$. (В этой формуле справа выполняются вычисления в шестнадцатеричной системе, результат — шестнадцатеричное число, однако оно неотличимо от десятичного и десятичным объявляется.)

Например,

$$x_{16} = 4_{16} \quad \Rightarrow \quad x_{10} = 4,$$

$$x_{16} = B_{16} \quad \Rightarrow \quad B_{16} + 6 = 11_{16} \quad \Rightarrow \quad x_{10} = 11_{10},$$

$$x_{16} = 12_{16} \quad \Rightarrow \quad 12_{16} + 6 = 18_{16} \quad \Rightarrow \quad x_{10} = 18_{10}.$$

Операции сложения и вычитания десятичных чисел выполняются поразрядно с парами цифр справа налево. При сложении возможен перенос единицы в старший разряд, а при вычитании — заем из старшего разряда. Так как числа записаны в двоично-десятичном коде, где десятичные цифры изображаются тетрадами, поразрядность выполнения операций означает, что они выполняются над тетрадами.

Если быть точным, операции выполняются над двоичными беззнаковыми числами и переносы и заемы могут возникать не только между тетрадами, но и между любыми разрядами чисел. Однако переносы внутри тетрад являются частью операций с тетрадами, и лишь переносы в пятый и девятый разряды определяют взаимодействие тетрад.

Сложение и вычитание выполняются с каждой парой тетрад в два этапа. На первом — тетрады складываются (вычитаются) как шестнадцатеричные числа. Так как результат получается шестнадцатеричным, на втором этапе он переводится в десятичную систему. Этот этап называют *десятичной коррекцией*.

Коррекция при сложении производится, если в результате выполнения первого этапа произошел перенос единицы в следующую тетраду или полученная тетрада не может быть интерпретирована как десятичная цифра. Заключается коррекция в прибавлении к тетраде результата числа 6. При вычитании коррекция производится, если был сделан заем единицы, и заключается в вычитании числа 6 из тетрады результата.

Рассмотрим примеры.

Пример 1. 12 + 19 = 31 (рис. 2.26).

Двоично-десятичные коды	1-й этап	2-й этап
$12 = 00010010_{2/10}$	00010010	00101011
	+	+
$19 = 00011001_{2/10}$	00011001	0110
$31 = 00110001_{2/10}$	00101011	00110001
	недопустимая комбинация	

Рис. 2.26

Пример 2. 38 + 49 = 87 (рис. 2.27).

Двоично-десятичные коды	1-й этап	2-й этап
$38 = 00111000_{2/10}$	00111000	10000001
	+	+
$49 = 01001001_{2/10}$	01001001	0110
$87 = 10000111_{2/10}$	10000001	10000111
	произошел перенос в старшую тетраду	

Рис. 2.27

Пример 3. 28 + 11 = 39 (рис. 2.28).

Двоично-десятичные коды	1-й этап	2-й этап
$28 = 00101000_{2/10}$	00101000	Переноса или недопустимой комбинации нет. 2-й этап не требуется
$11 = 00010001_{2/10}$	+ 00010001	
$39 = 00111001_{2/10}$	00111001	

Рис. 2.28

Пример 4. 54 – 29 = 25 (рис. 2.29).

Двоично-десятичные коды	1-й этап	2-й этап
$54 = 01010100_{2/10}$	01010100	00100101
$29 = 00101001_{2/10}$	– 00101001	– 0110
$25 = 00100101_{2/10}$	00100101	00100101
	произошел заем из старшей тетрады	

Рис. 2.29

Пример 5. 95 – 43 = 52 (рис. 2.30).

Двоично-десятичные коды	1-й этап	2-й этап
$95 = 10010101_{2/10}$	10010101	Заема не было, недопустимой комбинации нет. 2-й этап не требуется
$43 = 01000011_{2/10}$	– 01000011	
$52 = 01010010_{2/10}$	01010010	

Рис. 2.30

В некоторых компьютерах десятичные сложение и вычитание реализованы аппаратно, т. е. выполняются одной командой. В других таких операций нет,

но процессор предоставляет программисту специальные средства для реализации этих операций программно, т. е. несколькими командами.

Такими средствами являются признаки переноса в девятый и в пятый разряды, которые вырабатываются процессором при выполнении двоичного сложения, и признаки заема из девятого и пятого разрядов — при вычитании. Для запоминания этих признаков процессор имеет специальные одноразрядные регистры. Имеется также несколько специальных команд. Это команды: коррекция после сложения, коррекция после вычитания, сложение с переносом, вычитание с заемом.

Первые две команды используются после команд сложения и вычитания соответственно. Они выполняют второй этап десятичной арифметики сразу с двумя тетрадами, входящими в байт.

Вот как это происходит, например, при сложении:

1. По содержимому левой тетрады байта результата и признаку переноса в девятый разряд устанавливается необходимость в коррекции левого байта.
2. По содержимому правой тетрады байта результата и признаку переноса в пятый разряд устанавливается необходимость в коррекции правого байта.
3. При необходимости коррекции к байту результата прибавляется число 01100000, или 00000110, или 01100110.

Часто десятичные числа занимают больше одного байта. В этом случае байты складываются справа налево.

1. Складываются правые байты, выполняется десятичная коррекция.
2. Все последующие байты складываются с использованием операции "сложение с переносом".

Эта операция складывает байты и к сумме прибавляет единицу, если предыдущая операция выработала бит переноса.

Аналогично выполняется десятичное вычитание байтов и нескольких байтов.

ГЛАВА 3

Команды арифметико-логического типа и адресация

3.1. Принципиальная схема компьютера

3.1.1. Компьютер в целом

На рис. 3.1 представлена схема компьютера, которую можно считать классической.

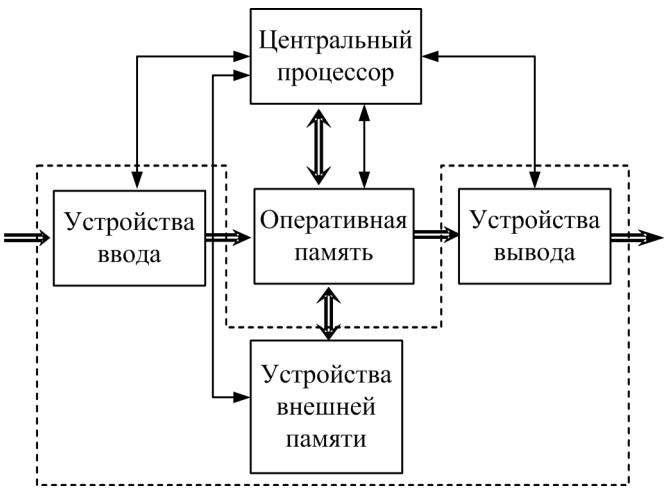


Рис. 3.1. Схема компьютера

В компьютере циркулируют два вида информации. Один вид — данные и перерабатывающие их программы. Второй — сигналы (их называют управляющими), которыми обмениваются между собой различные устройства

компьютера для координации действий. Направления передачи данных и программ показаны на рисунке двойными стрелками, управляющих сигналов — простыми.

Центральный процессор компьютера выполняет два вида работ: во-первых, перерабатывает, в соответствии с программой, данные и, во-вторых, координирует работу всех устройств. Управляющая информация передается в двух направлениях: устройства передают центральному процессору информацию о своем состоянии, центральный процессор передает им управляющие команды.

Оперативная память (используются также термины "основная память", "первичная память") предназначена для хранения данных и программ. В компьютерах принстонского или, иначе, фон-неймановского типа для данных и программ используется общая оперативная память. Существуют также компьютеры с отдельной памятью для данных и отдельной — для программ (гарвардского типа). Мы будем рассматривать наиболее распространенные компьютеры — фон-неймановского типа.

Устройства ввода предназначены для ввода программ и исходных данных. У одного компьютера обычно имеется несколько устройств ввода. Это, например, клавиатура, сканер, модем, мышь. *Устройства вывода* предназначены для вывода конечных результатов работы программ. Устройствами вывода являются принтер, монитор, модем и др. Часто используют общий термин — *устройства ввода/вывода*.

Устройства внешней памяти (ее называют также вторичной памятью, массовой памятью) предназначены для тех же целей, что и оперативная память, но они, во-первых, более медленные, хотя и более дешевые, во-вторых, с содержимым внешней памяти возможны лишь две операции: пересылка из оперативной памяти во внешнюю и пересылка из внешней памяти в оперативную. Нельзя, например, сложить два числа, хранящиеся во внешней памяти, или к числу из оперативной памяти прибавить число из внешней. Если такая операция требуется, нужно сначала переслать числа из внешней памяти в оперативную. Невозможно, также, выполнить команду, находящуюся во внешней памяти.

К устройствам внешней памяти относятся дисководы гибких и жестких дисков, дисководы оптических дисков, флэш-память, устройства на магнитных лентах. Надо заметить, что эти устройства можно, в определенном смысле, рассматривать и как устройства ввода/вывода, особенно сменные устройства. Их называют также устройствами хранения информации.

Устройства ввода/вывода и устройства внешней памяти иногда называют внешними устройствами. (На рис. 3.1 они обведены пунктирной линией.) Как

видно из рисунка, данные и программы из внешних устройств могут передаваться только через оперативную память.

Процессор и оперативная память являются ядром компьютера.

3.1.2. Память

Ячейка

Оперативная память (ОП) состоит из ячеек. Число их в разных машинах различно. В первых машинах это были тысячи, в современных — сотни миллионов. Ячейки ОП пронумерованы, начиная с нуля. Номера ячеек называют также их *адресами*. Все ячейки машины одинаковы. Любое данное или команду можно поместить в любую ячейку ОП. Время, которое затрачивается на запись информации в ячейку ОП, так же как и время, необходимое для чтения информации из ячейки, не зависит от ее номера. (Такую память называют *памятью прямого доступа*, в отличие, например, от магнитной ленты, на которой строится память последовательного доступа.)

Ячейка памяти обладает тремя важными свойствами.

- Содержимое ячейки сохраняется сколь угодно долго, пока процессор не выполнит команду, предписывающую изменить содержимое ячейки. В большинстве компьютеров содержимое ячейки теряется также при выключении питания.
- При записи информации в ячейку старое ее содержимое теряется и на новое содержимое никак не влияет. То есть фактически запись выполняется в два приема: сперва старое содержимое ячейки "стирается", а затем в пустую ячейку записывается новое содержимое.
- При пересылке из одной ячейки в другую содержимое первой не меняется. То есть происходит копирование содержимого первой ячейки во вторую.

Ячейки состоят из разрядов. Каждый разряд способен хранить один бит информации, т. е. значение одного разряда двоичного числа. Поэтому разряды ячейки также называют битами. Количество битов в ячейках разных компьютеров различно.

Ячейка является минимальной адресуемой единицей памяти. Это означает, что нет возможности заслать новое значение в один или несколько разрядов ячейки. Можно лишь целиком обновить ее содержимое. Точно так же невозможно прочитать содержимое нескольких разрядов, иначе как прочитав содержимое всей ячейки.

Команда

Минимальными единицами программы являются *команды*. В некоторых типах компьютеров принят термин "инструкция". Команды программы обычно

располагаются в последовательных ячейках памяти. В современных компьютерах команды, как, впрочем, и данные, могут располагаться не в одной, а в нескольких подряд расположенных ячейках. Но об этом позже. Пока будем считать, что каждая команда хранится в одной ячейке.

Команда предписывает процессору некоторое элементарное действие. Например, сложить два числа, хранящиеся в ячейках A и B, и результат заслать в ячейку C. Вопрос о том, какие именно действия следует считать элементарными, не имеет однозначного решения и для каждого типа компьютеров решается по-своему, хотя и сходно. Для обозначения совокупности всех возможных для данного типа компьютера команд или инструкций используются термины "система команд" и "набор инструкций".

Все команды можно разбить на три группы.

- Первая, самая многочисленная, — команды, перерабатывающие информацию. Их называют *командами арифметического типа*. Это команды арифметических действий, математической логики, сдвигов и некоторые другие.
- Вторая группа — *команды управления*. Эти команды сами не перерабатывают информацию, но управляют процессом переработки.
- Третью группу составляют *команды обмена*. Это команды чтения информации из внешних устройств в оперативную память и команды записи информации из оперативной памяти во внешние устройства. В большинстве компьютеров с внешними устройствами могут работать только команды этой группы.

Храниться команды могут как в оперативной, так и во внешней памяти, но процессор может исполнить лишь ту команду, которая находится в оперативной памяти.

Программа — это некоторая последовательность команд, записанных в последовательные ячейки памяти. После выполнения команды арифметического типа или команды обмена, как правило, выполняется следующая за ней команда. Команды управления могут нарушать эту естественную последовательность.

Локальная операция

Для описания процессов, происходящих в компьютере, используют понятие "локальная операция". В локальной операции указываются не переменные, как в обычной операции арифметического типа, а их адреса — места хранения переменных.

Мы будем пользоваться распространенным обозначением: $(A) = x$, которое читается так: "содержимое ячейки A равно x" или " x хранится по адресу

A". Пусть, например, требуется вычислить сумму $z = x + y$ и поместить z в ячейку C, причем $x = (A)$, $y = (B)$. Соответствующая локальная операция записывается в виде: $(A) + (B) \to C$ (иногда пишут $C \leftarrow (A) + (B)$). Эта локальная операция — трехместная, в ней фигурируют три адреса. Обобщенную локальную трехместную операцию будем записывать так: $(A)\,p\,(B) \to C$, где p — знак некоторой операции.

Большинству арифметических операций и операций математической логики соответствуют трехместные локальные операции. Однако существуют и такие операции, которые требуют двух адресов. Это, в первую очередь, сдвиги и пересылки, извлечение квадратного корня.

Например, `SLL(A)` $\to$ `B` (сдвиг логический влево) или $(A) \to B$. Обобщенная двухместная локальная операция: $p(A) \to B$.

Машинная команда (инструкция) — это локальная операция, записанная в кодах машины. Правила кодирования локальных операций в компьютерах разных типов различны. Чтобы сохранить общность, будем пользоваться в примерах записью фрагментов программ в виде последовательности локальных операций, не пользуясь какой-либо конкретной кодировкой.

Команда состоит из двух частей: операционной и адресной. В операционной части записывается код операции, в адресной — адреса операндов. Для записи кода операции и каждого из адресов в ячейке памяти отводятся определенные биты (совокупность этих битов называется *полем*), так что никаких разделителей (скобок и стрелок) уже не нужно.

Обычно трехадресная команда выглядит так, как показано на рис. 3.2.

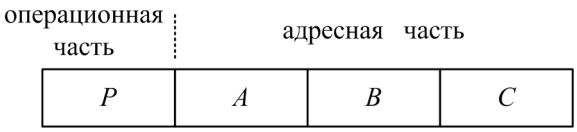


Рис. 3.2

Двухадресная команда короче (рис. 3.3).

Рис. 3.3

В связи с тем, что существуют как трехадресные, так и двухадресные локальные операции арифметического типа, перед конструктором машины встает следующая проблема. Представим на минуту, что мы строим машину только для трехадресных команд. Естественное решение — все команды кодируются единообразно, длина всех команд одинакова, адресная часть, как и операционная, имеет одну длину у всех команд. А теперь мы хотим переделать нашу машину таким образом, чтобы в ней могли выполняться также команды, реализующие двухместные локальные операции. Есть два решения проблемы.

- Ввести, наряду с трехадресными, двухадресные команды. Тогда придется существенно усложнить и память и процессор. Память должна состоять из ячеек переменной длины, а процессор должен различать и по-разному декодировать команды с разной адресностью. Это удорожает процессор и увеличивает время выполнения команды.
- Унифицировать команды, сделав их все, например, формально трехадресными.

Трехадресная команда пересылки может иметь, например, вид, как на рис. 3.4.

Рис. 3.4

Символ "~" означает, что в качестве второго адреса можно указать любое число. Это число никак не повлияет на выполнение операции. Обычно в этой позиции пишут 0.

Очевидно, что унификация снижает эффективность использования памяти, т. к. часть ее, как в командах пересылки, пропадает зря. Но зато процессор может обрабатывать все команды единообразно.

3.1.3. Процессор

На рис. 3.5 приведена схема процессора и оперативной памяти. Это логическая схема, удобная для понимания принципов работы и взаимодействия процессора и оперативной памяти. Физически процессоры разных машин устроены по-разному, похоже на нашу схему, но значительно сложнее.

На рисунке изображены:

- устройство управления (УУ), координирующее работу всех блоков компьютера;

- ряд регистров — указатель команд (УК), регистр команд (РК), регистр первого операнда ($r1$), регистр второго операнда ($r2$), регистр результата ($r3$). Эти регистры — запоминающие устройства, подобные ячейкам памяти, но гораздо более быстрые;
- арифметико-логическое устройство (АЛУ). Его можно представлять в виде набора отдельных блоков, каждый из которых выполняет одну из операций, входящих в систему команд. Система команд обычно насчитывает одну-две сотни команд;
- оперативная память (ОП). В современных машинах — сотни миллионов ячеек.

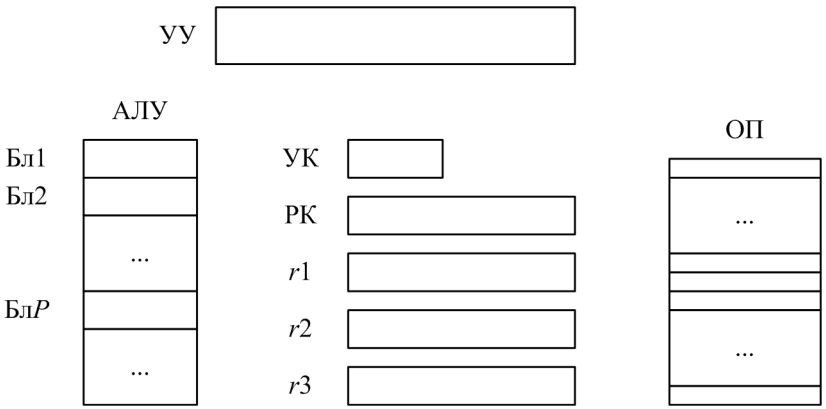


Рис. 3.5. Схема процессора

Операции выполняются всегда с содержимым регистров $r1$ и $r2$, а результат засылается в $r3$. УУ, АЛУ и все названные регистры составляют процессор.

Процессор способен выполнять только команду, находящуюся в настоящий момент в РК. Для этого команды поочередно вызываются в РК. Адрес команды, которую нужно вызвать, хранится в УК.

3.2. Основные этапы выполнения команды арифметического типа

Машины можно классифицировать по адресности — трехадресные, двухадресные, одноадресные. Первоначально строились именно такие машины. Однако сегодня, в большинстве случаев, об адресности машин можно говорить лишь условно. Например, компьютеры на базе процессоров Intel имеют трех-, двух-, одноадресные и даже безадресные команды. И все же по большинству команд эту машину можно назвать двухадресной.

Далее мы рассмотрим выполнение команд в машинах разной адресности, имея в виду, что в реальных машинах процессор перед исполнением команды определяет ее адресность, а затем выполняет так, как выполнила бы ее машина соответствующей адресности.

3.2.1. Трехадресная машина

Представим, что мы остановили компьютер, выполняющий некоторую программу, в самый рядовой момент его работы. Только что закончилось исполнение какой-то команды и процессор готов к выполнению следующей. В регистре — указателе команд как раз находится адрес этой команды — A. (Содержимое остальных регистров никакой роли не играет.)

Первый этап выполнения команды показан на рис. 3.6.

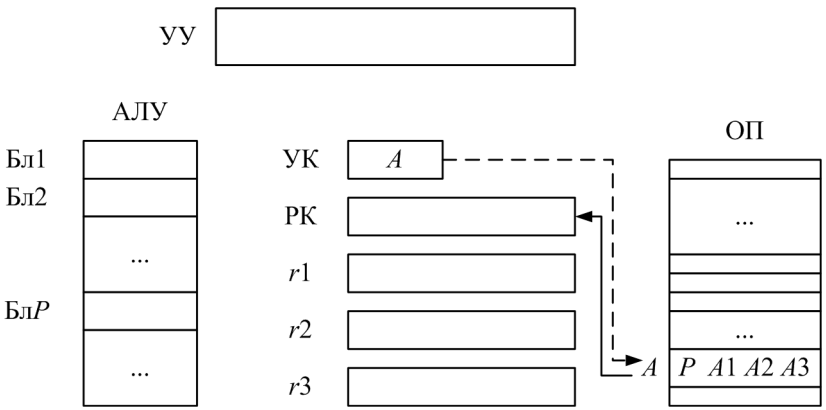


Рис. 3.6. Первый этап выполнения команды

Все начинается с того, что УУ извлекает из ОП команду, хранящуюся по адресу, хранящемуся в УК, и пересылает ее в РК.

После того как команда помещена в РК, содержимое УК увеличивается на 1, — процессор подготавливается к вызову следующей команды.

Следующий этап — дешифрирование кода операции. УУ читает код операции команды, находящейся в РК, и активизирует соответствующий блок (рис. 3.7).

Затем производится вызов операндов: УУ прочитывает адресную часть команды, находящейся в РК, и определяет адреса. Содержимое этих адресов последовательно вызывается в $r1$ и $r2$. (рис. 3.8.)

Следующий этап. Блок P выполняет операцию с содержимым регистров $r1$ и $r2$ и результат операции помещает в $r3$ (рис. 3.9).

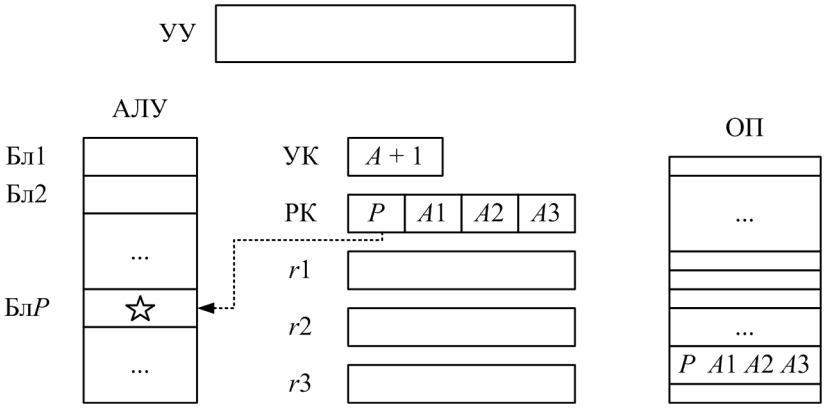


Рис. 3.7. Этап дешифрирования кода операции

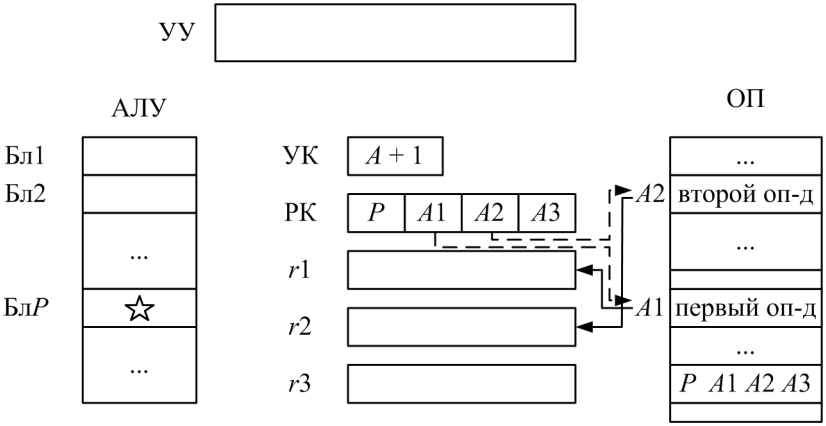


Рис. 3.8. Чтение данных из памяти

Последний этап — рис. 3.10. Содержимое $r3$ засылается в память по адресу $A3$.

Процессор работает по принципу шарманки. Совершив полный оборот, шарманка снова играет ту же мелодию. Процессор, сделав последний шаг в исполнении команды, возвращается к первому шагу, но уже с новым содержимым УК.

Запишем кратко основные этапы исполнения команды арифметического типа.

1. Содержимое содержимого УК переслать в РК {еще короче: (УК) → РК}.
2. Продвинуть содержимое УК {(УК) + 1 → УК}.

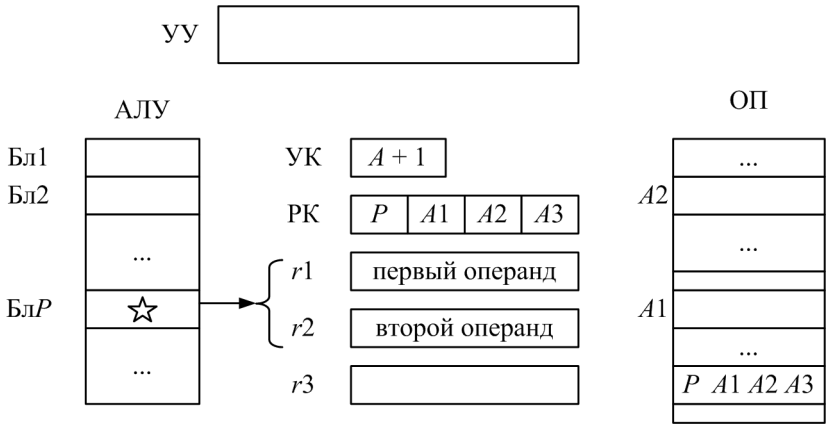


Рис. 3.9. Этап непосредственного выполнения операции

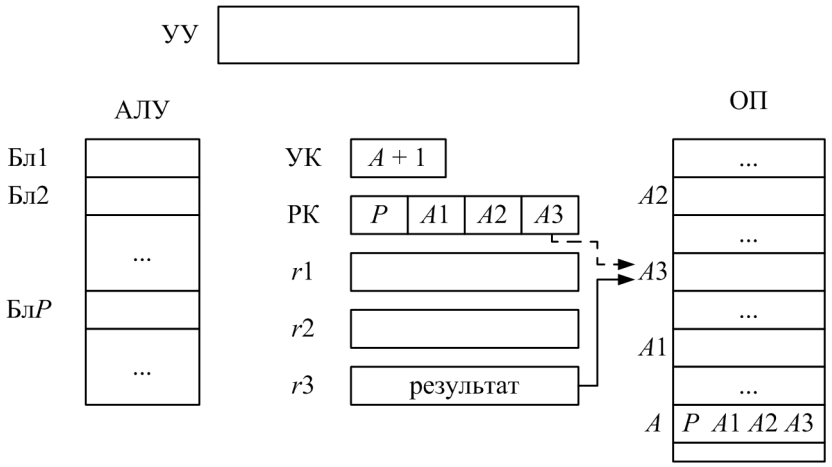


Рис. 3.10. Этап записи результата

3. Дешифрировать код операции — активизировать блок P.
4. Содержимое ячейки $A1$ — переменную $x1$ переслать в $r1$ $\{(A1) \rightarrow r1\}$.
5. Содержимое ячейки $A2$ — переменную $x2$ переслать в $r2$ $\{(A2) \rightarrow r2\}$.
6. Выполнить операцию с содержимым регистров $r1$ и $r2$, результат поместить в регистр $r3$ $\{(r1)p(r2) \rightarrow r3\}$.
7. Переслать результат в память $\{(r3) \rightarrow A3\}$.
8. Перейти к выполнению пункта 1.

Заметим, что при такой организации машины АЛУ работает в точности как калькулятор. Ему нет дела до того, что существует оперативная память, а уж тем более до внешних устройств. Он выполняет указанную ему операцию над содержимым двух фиксированных регистров и результат помещает всегда в один и тот же регистр. Устройство управления работает как коммутатор — соединяет нужные ячейки и нужный блок АЛУ с регистрами.

3.2.2. Двухадресные машины

В памяти компьютера, наряду с программой и исходными данными, должны храниться промежуточные и конечные результаты. Ячейки, используемые для хранения промежуточных результатов, называют *рабочими*. (Подчеркнем еще раз: все ячейки одинаковы, любая может быть использована для хранения команды, исходного данного или как рабочая.) Выделим среди всех данных две особые группы.

1. Данные, которые используются в алгоритмах только один раз. Это могут быть исходные данные или промежуточные результаты.
2. Данные, которые используются немедленно после их вычисления. Это могут быть только промежуточные результаты.

Данных, относящихся к одной из этих групп, или к обеим сразу, очень много, может быть даже большинство, и в конструкции компьютеров это обстоятельство используется для повышения их эффективности.

Двухадресные машины первого типа

Конструкция этих машин опирается на предположение, что данные, которые используются в алгоритмах только один раз, составляют значительную часть всех данных. Рассмотрим пример.

Пример 1.

$$W = \frac{U}{V}, \quad \text{где } U = X_1 \times X_2 + X_3 \times X_4,\ V = X_1 \times X_2 + X_3 \times X_5.$$

Исходные данные X_4 и X_5 используются по одному разу, X_3 — два раза. X_1 и X_2, хотя и встречаются по два раза в формуле, могут быть использованы по одному разу для вычисления промежуточной величины $X_1 \times X_2$, которая встречается два раза. Промежуточные результаты $X_3 \times X_4$, $X_3 \times X_5$, U и V используются по одному разу.

Обычно промежуточных данных, используемых один раз, настолько много, что можно получить серьезную экономию памяти, многократно пользуясь одними и теми же рабочими ячейками.

Предположим $X_i = (A_i)$. Для вычисления W из примера 1 можно использовать такой алгоритм:

1. $(A1) \times (A2) \to B1$.
2. $(A3) \times (A4) \to B2$.
3. $(B1) + (B2) \to B2$,
4. $(A3) \times (A5) \to B3$.
5. $(B1) + (B3) \to B1$.
6. $(B2) / (B1) \to B1$.

Ячейки $B1$, $B2$, $B3$ — рабочие. Нетрудно видеть, что за счет многократного использования ячеек $B1$ и $B2$ число рабочих ячеек сокращено вдвое. Можно увеличить экономию, если использовать в качестве рабочих ячейки, содержащие исходные данные.

1. $(A1) \times (A2) \to A1$.
2. $(A3) \times (A4) \to A4$.
3. $(A1) + (A4) \to A4$.
4. $(A3) \times (A5) \to A5$.
5. $(A5) + (A1) \to A1$.
6. $(A4) / (A1) \to A1$.

Последнее решение замечательно еще и тем, что во всех его трехместных локальных операциях используются лишь два различных адреса. Исследуем вопрос, всегда ли такое возможно. Заметим сразу, что сомнения возникают лишь в связи с исходными данными или промежуточными результатами, используемыми более одного раза. Если промежуточный результат или исходное данное нужно только один раз, операция, использующая такое данное, может помещать результат на место этого данного. Рассмотрим пример.

Пример 2.

$$W = \frac{U}{V}, \quad \text{где } U = X_1 \times X_2 + X_3 \times X_4, \; V = X_1 \times X_4 + X_3 \times X_5.$$

На первый взгляд кажется, что, по крайней мере, в одной локальной операции необходимо использовать три различных адреса: произведение $X_3 \times X_4$ нельзя поместить ни в ячейку $A3$, ни в ячейку $A4$, т. к. хранящиеся там переменные X_3 и X_4 понадобятся в дальнейшем. Однако за счет изменения порядка действий можно решить эту проблему.

1. $(A1)\times(A2)\to A2$.
2. $(A3)\times(A5)\to A5$.
3. $(A4)\times(A3)\to A3$.
4. $(A3)+(A2)\to A2$.
5. $(A4)\times(A1)\to A1$.
6. $(A5)+(A1)\to A1$.
7. $(A2)/(A1)\to A1$.

В следующем примере все переменные входят в формулу более одного раза, поэтому требуется использовать специальный прием.

Пример 3.

$$W=\frac{U}{V},\quad \text{где}\ \ U=X_1\times X_2+X_3\times X_4,\ \ V=X_3\times X_2+X_1\times X_4.$$

Выполним, для начала, в качестве пункта 0 вспомогательную операцию $(A2)\to A5$ и далее все семь пунктов предыдущего примера. Задача решена.

Этот прием является универсальным. Изготовив, с помощью пересылок, нужное количество копий переменных, мы можем затем в каждой локальной операции уничтожать по одной копии, засылая на ее место промежуточный результат.

Таким образом, для всех задач существует возможность в каждой локальной операции использовать только два различных адреса. Это позволяет построить двухадресную машину, т. е. машину, все команды которой двухадресные. Разумеется, двухадресные локальные операции легко кодируются двухадресной командой. Трехадресную локальную операцию можно втиснуть в рамки двухадресной команды, только используя прием подразумевания. Построено много двухадресных машин, в которых подразумевается, что адрес результата всегда совпадает с одним из двух первых адресов. Именно такие двухадресные машины, будем называть машинами первого типа.

Будем считать, например, что результат заносится по второму адресу команды. Тогда двухадресная команда "*P A B*" выполняется так: $p(A)\to B$, если команда соответствует двухместной локальной операции и $(A)p(B)\to B$, — если трехместной.

Преимуществом таких машин перед трехадресными является экономия памяти, занимаемой программой. Правда, эффект от уменьшения длины команд снижается из-за появления дополнительных команд копирования данных, но, как правило, ненамного.

Процесс выполнения команд в двухадресной машине первого типа отличается от аналогичного процесса в трехадресной машине незначительно: запись результата в память производится не по третьему, а по второму адресу (седьмой этап), т. е. процессор выполняет ровно тот же объем работ при выполнении каждой команды. Однако из-за команд копирования общее время выполнения программы несколько возрастает.

Краткая запись основных этапов выполнения команд арифметического типа в двухадресных машинах первого типа.

1. Содержимое содержимого УК переслать в РК $\{((\text{УК})) \rightarrow \text{РК}\}$.
2. Продвинуть содержимое УК $\{(\text{УК}) + 1 \rightarrow \text{УК}\}$.
3. Дешифрировать код операции — активизировать блок P.
4. Содержимое ячейки $A1$ — переменную $x1$ переслать в $r1$ $\{(A1) \rightarrow r1\}$.
5. Содержимое ячейки $A2$ — переменную $x2$ переслать в $r2$ $\{(A2) \rightarrow r2\}$.
6. Выполнить операцию с содержимым регистров $r1$ и $r2$, результат поместить в регистр $r3$ $\{(r1)\,p(r2) \rightarrow r3\}$.
7. Переслать результат в память $\{(r3) \rightarrow A2\}$.
8. Перейти к выполнению пункта 1.

Двухадресные машины второго типа

Организация машин второго типа опирается на то обстоятельство, что некоторые промежуточные результаты могут быть использованы немедленно. Последний этап выполнения команды делается в этом случае явно излишним. Процессор отсылает в память результат операции, который при выполнении следующей команды будет снова вызван из памяти. В двухадресных машинах второго типа в качестве одного из адресов трехместных операций используется регистр результата ($r3$). Его называют также аккумулятором (A) или сумматором (S) (мы будем пользоваться последним обозначением). Можно сказать, что это — трехадресные машины с одним фиксированным адресом.

В таких машинах процессор, в зависимости от кода операции, выполняет команды арифметического типа по одной из двух схем. В соответствии с этим в команде появляется еще одно поле — поле признаков типа адреса σ (рис. 3.11).

Код операции P называет выполняемую операцию, признак σ — схему, по которой она должна быть выполнена. Так как схемы всего две, поле признаков занимает один бит.

Основные этапы выполнения команд в двухадресных машинах второго типа изображены на рис. 3.12.

Рис. 3.11

При $\sigma = 0$ операнды извлекаются из памяти, а результат в память не записывается, т. е. остается в S. Кратко: $(A1)\,p\,(A2) \to S$. Если $\sigma = 1$, то лишь один операнд извлекается из памяти, а второй — из S. Результат же помещается в память. Кратко: $(A1)\,p\,(S) \to A2$.

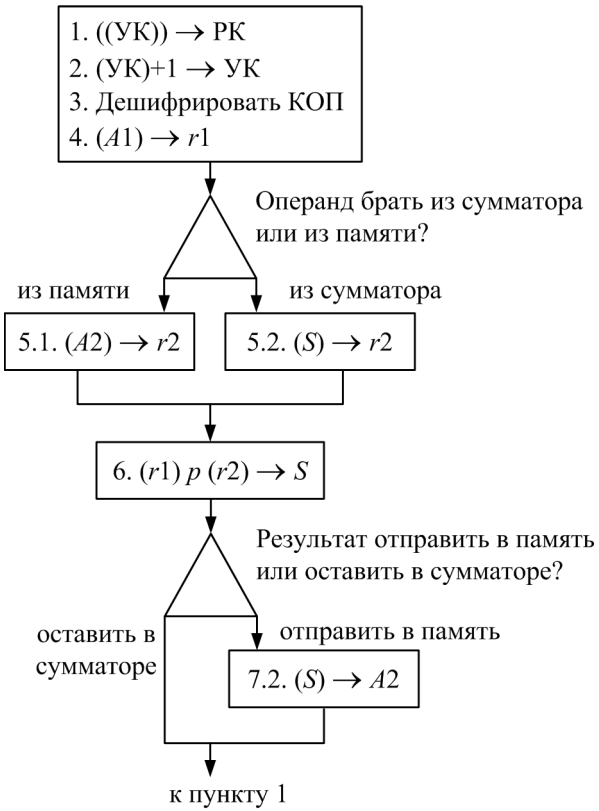


Рис. 3.12. Основные этапы выполнения команд в двухадресных машинах второго типа

Первые четыре этапа выполнения команды и шестой в двухадресной машине второго типа совпадают с теми, что описаны для трехадресной машины.

Главное достоинство двухадресной машины второго типа — меньшее время выполнения каждой команды. Здесь производится всего три обращения к памяти на команду. Первое — при вызове команды в РК, второе — при вызове первого операнда, третье — при вызове второго операнда (по схеме 0) или при засылке результата (по схеме 1). Пересылка из S в $r2$, возникающая при выполнении команды по схеме 1, производится гораздо быстрее обращения к памяти и существенного влияния на время выполнения команды не оказывает. В трехадресной машине и в двухадресной машине первого типа производится четыре обращения к памяти на команду.

Имеется еще одно, формальное отличие. В двухадресной машине второго типа, в отличие от трехадресной машины и двухадресной машины первого типа, на результат операции влияет не только содержимое памяти и УК. При выполнении операции по схеме 1 существенным оказывается также содержимое S, т. е. S является не просто одним из внутренних устройств процессора, но выступает как устройство памяти, дополняющее ОП.

Рассмотрим примеры вычислений в двухадресной машине второго типа.

Пример 4.

$$W = \frac{U}{V}, \quad \text{где } U = X_1 + X_2 \times X_3,\ V = X_4 + X_5 \times X_6.$$

Пусть, как и раньше, $(Ai) = X_{i1}$. Алгоритм решения можно записать как такую последовательность локальных операций:

1. $(A2) \times (A3) \to S$.
2. $(A1) + (S) \to A1$.
3. $(A5) \times (A6) \to S$.
4. $(A4) + (S) \to A4$.
5. $(A1) / (A4) \to S$.

Очевидно, что команды, соответствующие локальным операциям 1, 3 и 5, выполняются по схеме 0, локальным операциям 2 и 4 — по схеме 1. Это пример удачного алгоритма. Здесь оказалось возможным все результаты, полученные по схеме 0, немедленно использовать. Это случается далеко не всегда.

Пример 5.

$$W = \frac{U}{V}, \quad \text{где } U = X_1 \times X_2 + X_3 \times X_4,\ V = X_1 \times X_2 + X_3 \times X_5.$$

1. $(A1)\times(A2)\to S$.
2. $(S)\to A1$.
3. $(A4)\times(A3)\to S$.
4. $(A1)+(S)\to A2$.
5. $(A3)\times(A5)\to S$.
6. $(A1)+(S)\to A1$,
7. $(A2)/(A1)\to S$

Вторая операция этого алгоритма оказалась одноадресной.

В этой машине существуют три разновидности операций пересылки: "S — память", "память — S", "память — память". Первые две — одноадресные. Так как все команды машины формально двухадресные, в этих командах на месте второго адреса можно поставить любое число, — оно не повлияет на результат.

Шестая операция — обычная двухадресная. Просто два адреса совпадают.

В этом алгоритме не удалось немедленно использовать все промежуточные результаты, полученные по схеме 0. Результат первой операции пришлось резервировать в специальной рабочей ячейке. Нетрудно видеть, что две первых операции по своему результату эквивалентны одной операции трехадресной машины. Однако время выполнения этих двух операций несколько больше (два вызова операндов, одна отсылка результата, как и в трехадресной машине, но два вызова команд). Да и памяти две двухадресных команды занимают больше (два кода операции и четыре адреса).

И все же, в целом, этот алгоритм выполняется быстрее, чем соответствующий алгоритм для трехадресной машины (см. пример 1). Здесь количество обращений к памяти $3 \times 6 + 2 = 20$, в трехадресной машине — $4 \times 6 = 24$.

В памяти здесь хранится 7 кодов операции и 14 адресов. В трехадресной машине — 6 кодов операции и 18 адресов.

Рассмотрим еще менее благоприятный случай.

Пример 6.

$$W=\frac{U}{V},\quad \text{где}\ \ U=X_1\times X_2+X_3\times X_4,\ \ V=X_3\times X_5+X_1\times X_4.$$

Будем считать команды, вычисляющие W, фрагментом программы. При этом положим, что результат немедленно использоваться не будет, а значит, его нельзя оставить в $r3$.

1. $(A1)\times(A2)\to S$.
2. $(S)\to A2$.
3. $(A4)\times(A3)\to S$.
4. $(A2)+(S)\to A2$.
5. $(A3)\times(A5)\to S$.
6. $(S)\to A3$.
7. $(A1)\times(A4)\to S$.
8. $(A3)+(S)\to A3$.
9. $(A2)/(A3)\to S$.
10. $(S)\to A1$.

Число обращений к памяти здесь $3 \times 7 + 2 \times 3 = 27$, в алгоритме для трехадресной машины (см. пример 2) — $4 \times 7 = 28$ обращений к памяти.

Примеры 3, 4 и 5 иллюстрируют тот факт, что в двухадресной машине второго типа задачи решаются, как правило, быстрее, чем в трехадресной.

Памяти для программ в этой машине требуется меньше, чем в трехадресной. Однако здесь эффект меньше, чем в двухадресной машине первого типа, т. к. выигрыш достигается только в том случае, если немедленно используемое данное является также однократно используемым.

3.2.3. Одноадресные машины

Идея экономии числа обращений к памяти за счет уменьшения количества адресов в команде находит свое естественное продолжение в одноадресных машинах. В командах этих машин адресная часть содержит только один адрес. Каждая трехместная локальная операция может быть выполнена по любой из двух схем: $(A)p(S)\to S$ (схема 0) или $(A)p(S)\to A$ (схема 1). Команда, как и в двухадресной машине, содержит признак σ — номер схемы. Здесь совмещаются идеи организации двухадресных машин первого и второго типов. Как в машинах первого типа, третий адрес совпадает с одним из двух первых адресов, как в машинах второго типа одним из адресов является S. Можно также считать одноадресную машину двухадресной с одним фиксированным адресом. Вот как выглядят основные этапы выполнения команд в одноадресных машинах (рис. 3.13).

Рассмотрим пример, иллюстрирующий работу одноадресной машины.

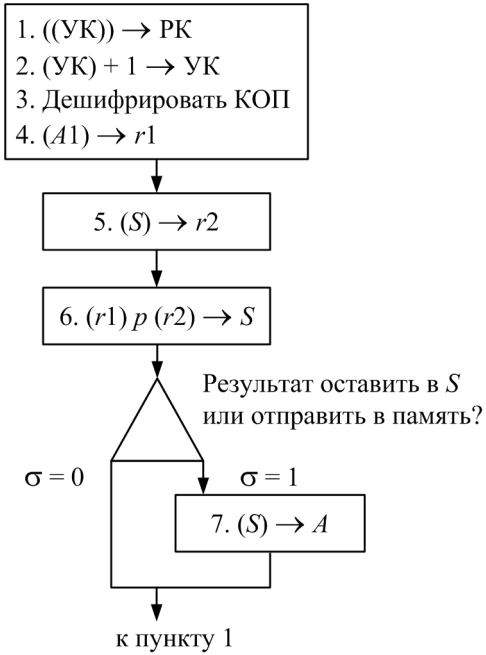


Рис. 3.13. Основные этапы выполнения команд в одноадресных машинах

Пример 7.

$$W = \frac{U}{V}, \quad \text{где } U = X_1 \times X_2 + X_3 \times X_4,\ V = X_1 \times X_2 + X_3 \times X_5.$$

1. $(A1) \to S$.
2. $(A2) \times (S) \to A2$.
3. $(A3) \to S$.
4. $(A4) \times (S) \to S$.
5. $(A2) + (S) \to S$.
6. $(S) \to B1$.
7. $(A3) \to S$.
8. $(A5) \times (S) \to S$.
9. $(A2) + (S) \to S$.
10. $(B1) / (S) \to B1$.

Здесь операции 1, 3 и 7 — подготовительные. Они действуют в паре с операциями 2, 4 и 8 соответственно. В трехадресной машине каждой из этих пар соответствует по одной команде. Операция 6 сохраняет промежуточный результат, который не может быть использован немедленно.

Число обращений к памяти — 22. (Вспомним, что в трехадресной машине и двухадресной первого типа — 24, второго — 20.)

3.2.4. Сравнение машин разной адресности

В некоторых случаях одноадресная машина оказывается поразительно эффективной. Сравним число обращений к памяти и количество команд в алгоритмах для различных типов машин при решении задачи суммирования элементов массива.

Пример 8.

$$W = X_1 + X_2 + \ldots + X_N .$$

Алгоритм для одноадресной машины:

1. $(A1) \to S$.
2. $(A2) + (S) \to S$.
3. $(A3) + (S) \to S$.

...

N. $(AN) + (S) \to S$.

Здесь N одноадресных команд и $2N$ обращений к памяти.

Для двухадресной машины первого типа:

1. $(A2) + (A1) \to A1$.
2. $(A3) + (A1) \to A1$.
3. $(A4) + (A1) \to A1$.

...

$N-1$. $(AN) + (A1) \to A1$.

В этом случае необходима $N-1$ двухадресная команда и $4(N-1)$ обращений к памяти. Очевидно, что алгоритм для трехадресной машины будет точно таким же, только длина программы больше — все команды будут трехадресными.

Для двухадресной машины второго типа:

1. $(A1)+(A2)\to S$.
2. $(A3)+(S)\to B1$.
3. $(B1)+(A4)\to S$.

...

$N-1$. $(B1)+(AN)\to S$.

Последняя операция имеет такой вид, если N четно. В противном случае:

$N-1$. $(AN)+(S)\to B1$.

Здесь тоже $N-1$ двухадресная команда, но $3(N-1)$ обращений к памяти.

Теперь пример противоположного сорта. Пусть в памяти хранятся два вектора: X и Y.. Причем

$$X_1=(A1),\ X_2=(A2),\ ...,\ X_N=(AN);\ Y_1=(B1),\ Y_2=(B2),\ ...,\ Y_N=(BN).$$

И пусть требуется вычислить сумму Z этих векторов и разместить в памяти следующим образом:

$$Z_1=(C1),\ Z_2=(C2),\ ...,\ Z_N=(CN).$$

Очевидно, трехадресная машина потребует N команд вида $(Ai)+(Bi)\to Ci$ и $4N$ обращений к памяти.

Двухадресной машине первого типа потребуется по две команды на каждое сложение:

1. $(Bi)\to Ci$.
2. $(Ai)+(Ci)\to Ci$.

Итого, $2N$ команд и $7N$ обращений к памяти.

Двухадресной машине второго типа — также по две команды на каждое сложение:

1. $(Bi)\to S$.
2. $(Ai)+(S)\to Ci$.

Но общее число обращений к памяти меньше — $5N$.

Одноадресной машине потребуются уже три команды на каждое сложение:

1. $(Bi)\to S$.
2. $(Ai)+(S)\to S$.
3. $(S)\to Ci$.

Общее число обращений к памяти — $6N$.

Вывод из приведенных примеров очевиден: нельзя однозначно ответить на вопрос, какой тип машин эффективнее. Для каждого типа существует область, в которой эффективность данного типа машин выше, иногда намного, чем других.

3.3. Машины с регистрами общего назначения

В двухадресных машинах второго типа регистр результата применялся как расширение оперативной памяти для хранения промежуточных результатов. Однако содержимое *S* требовалось немедленно использовать. Если это было невозможно или содержимое *S* должно было использоваться неоднократно, необходимо было создавать копию, что снижало эффективность использования *S*. Естественно напрашивалась мысль использовать несколько регистров для размещения промежуточных результатов. Имея несколько регистров, можно, если не избежать копирования, то сделать эту операцию достаточно редкой. Кроме того, появляется возможность ввести команды, в которых участвуют одновременно два регистра, уменьшая тем самым число обращений к памяти. Правда, если раньше регистр результата, как один из адресов подразумевался, то теперь придется явно называть регистры по номерам.

Доступные программисту регистры делятся на специализированные, например регистры для данных с плавающей точкой, и регистры общего назначения (РОН). В этом разделе рассматриваются только РОН. Своим названием РОН обязаны тому обстоятельству, что все они совершенно одинаковы, разумеется, не считая номера, и среди них не существует какой-либо специализации. Регистр результата, который по-прежнему является частью процессора, в число РОН не входит. То есть результат операции в *S* никогда не оставляется.

На рис. 3.14 представлена схема компьютера с РОН. Здесь не показаны пути прохождения управляющих сигналов. Они идут от центрального процессора ко всем остальным блокам и от них к центральному процессору. Зато отдельно показаны пути прохождения команд (двойные стрелки) и данных (простые стрелки). Из схемы видно, что РОН, являясь продолжением оперативной памяти, все же от нее отличаются: команды в РОН не попадают. Часто РОН считают частью центрального процессора.

В принципе, в двухадресной машине с регистрами общего назначения могут использоваться команды четырех типов (в трехадресной их еще больше, но мы будем рассматривать только двухадресную) — рис. 3.15.

Команды первого типа называют командами типа "регистр—регистр", второго и третьего — "регистр — память" и "память — регистр", четвертого — "память — память".

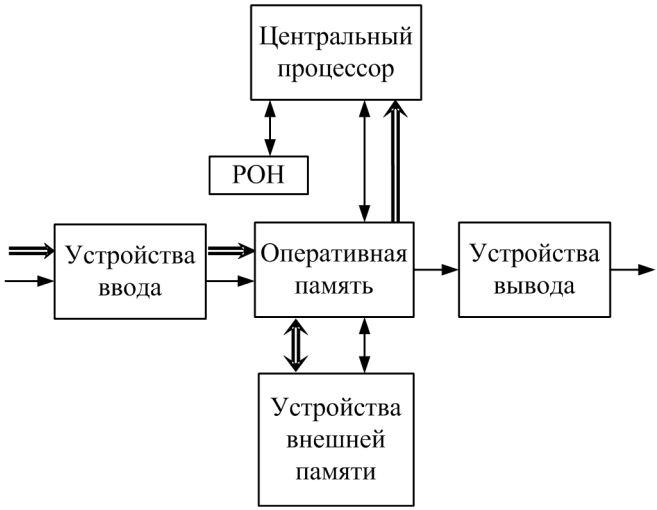


Рис. 3.14. Схема компьютера с РОН

P	*R*1	*R*2
P	*R*1	*A*2
P	*A*1	*R*2
P	*A*1	*A*2

Рис. 3.15. Типы команд в двухадресной машине

В машинах с регистровой адресацией возникает проблема: командам типа "регистр — регистр" нужна короткая ячейка, командам типа "регистр — память" и "память — регистр" — длинная, а командам "память — память" — еще длиннее.

Здесь возможны два решения: иметь систему команд одной длины, как-то приспособив ее для обращения и к памяти, и к регистрам, или создать команды разной длины.

3.3.1. Система команд фиксированной длины

Наиболее простой вариант — длинная команда, такая как изображена на рис. 3.16.

В этой команде вместо каждого из адресов памяти может быть записан номер регистра общего назначения. В варианте с длинной командой логично ис-

пользовать общее адресное пространство для ячеек памяти и для РОН. В этом случае нумерация регистров и ячеек общая. Если, например, имеется регистр № 1, то ячейки № 1 не существует. В такой машине нет специальных команд для работы с регистрами. Можно считать, что регистров и вовсе нет. Просто первые ячейки очень быстрые. Главное достоинство таких машин — единообразие команд.

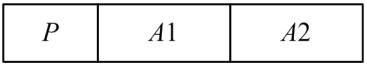


Рис. 3.16

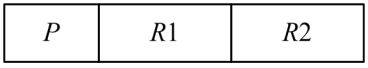


Рис. 3.17

Второй вариант команды фиксированной длины — короткая команда (рис. 3.17).

Вариант с короткой командой использует косвенную и непосредственную адресацию и будет обсуждаться в *разд. 3.4*.

Третий вариант — сверхдлинная команда. Это наиболее современное решение. Оно обсуждается в *главе 7*.

3.3.2. Система команд разной длины. Байтовая память

Это наиболее популярное направление развития системы команд. Естественно возникают три варианта длины команды (рис. 3.18).

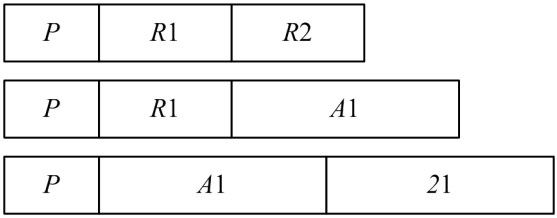


Рис. 3.18. Варианты команд с регистрами

Это усложняет работу процессора, но зато приводит к экономному использованию памяти, уменьшает длину программы. Надо иметь в виду, что уменьшение длины программы ценно не только само по себе. Более короткие команды быстрее вызываются из памяти в процессор, что приводит к уменьшению времени выполнения программ.

Для реализации системы команд разной длины нужна ячейка переменной длины.

Уже в первых компьютерах обнаружилась противоречивость требований к длине ячейки. Первое противоречие — между требованиями со стороны целых чисел и чисел с плавающей точкой.

Числа с плавающей точкой обычно требуют для записи больше разрядов, чем целые. Ведь, если припомнить, в математических формулах часто встречаются различные константы, такие как 0, 1, 6 и т. д. Вряд ли вам вспомнится формула с пятизначной целой константой. Не то чтобы их не было совсем, но они крайне редки. Таким образом, если длина ячейки достаточна для записи с нужной точностью чисел с плавающей точкой, то, записывая в нее короткие целые, мы напрасно тратим память.

Первоначально компьютеры в основном ориентировались на инженерные расчеты. А так как целые числа в инженерных формулах встречаются гораздо реже чисел с плавающей точкой, большинство компьютеров имело длинную ячейку. Такая ячейка была удобна и для записи в нее команд.

Однако при создании двух- и особенно одноадресных машин появляется новое противоречие — между длиной команд и длиной чисел. Если длина трехадресной команды была примерно такой, какая требовалась для размещения числа с плавающей точкой, то одноадресная команда гораздо короче.

Одно из первых решений этой проблемы — длинная ячейка, достаточная для хранения одного числа с плавающей точкой или двух одноадресных команд. Процессор в этой машине всегда выбирает в РК содержимое всей ячейки, и команды выполняются по такой схеме:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Выполнить левую команду.
4. Выполнить правую команду.
5. Перейти к выполнению следующей команды.

Недостатком такого решения является невозможность передачи управления сразу правой команде. Кроме того, если левая команда является командой передачи управления, правую половину ячейки нельзя использовать.

Тем временем происходили изменения в области применения компьютеров. Первоначально они применялись главным образом для научно-технических расчетов и в основном выполняли операции над числами с плавающей точкой. Постепенно, с ростом числа компьютеров и удешевлением машинного времени, компьютеры все больше стали применяться для хранения и переработки текстовой или, лучше сказать, символьной информации. Сначала для кодирования символов использовали 6 бит. Если надо было запомнить не много символов, то их записывали по одному в ячейку. В противном случае

приходилось, для экономии памяти, упаковывать в ячейку по несколько символов. Однако обработка упакованной информации требует значительно больше машинного времени.

Команды, адресующиеся к регистрам и к ячейкам памяти, числа с плавающей точкой и целые, символы — все эти единицы информации предъявляют различные требования к длине ячейки. Обострение противоречий сделало очевидной необходимость отказа от единого размера ячейки. К этому времени прогресс в области электроники сделал возможным поставить вопрос об усложнении процессора, сохраняя стоимость компьютера в разумных пределах.

Требование предоставить возможность записывать команды и данные в наиболее подходящие для них ячейки, может быть удовлетворено двумя путями: физическим — приспособлением ячеек к типам хранимой информации и программным.

Первый путь требует отказа от фон-неймановской архитектуры и создания различной памяти для команд и данных. Это решение не получило широкого распространения.

Второй путь — программное регулирование длины ячеек. То есть, как и прежде, имеются физические ячейки памяти, но для команд и данных используются логические ячейки, состоящие из одной или нескольких подряд расположенных физических ячеек. Адресом такой логической ячейки является наименьший из адресов входящих в нее физических ячеек или, как говорят, адрес самой левой ячейки.

Таким образом, физическая ячейка определяет размер кванта, на который может меняться длина логической ячейки. Длина физической ячейки определяется следующими соображениями. При уменьшении длины физической ячейки, с одной стороны, гибче использование памяти, меньше ее потери. С другой стороны, увеличивается число вариантов длины, что приводит к усложнению работы процессора. Кроме того, увеличивается количество физических адресов (при одинаковой емкости памяти в битах), следовательно, увеличивается длина команд, обращающихся к ячейкам памяти. Логично выбрать длину физической ячейки так, чтобы она соответствовала некоторой единице информации или вмещала бы целое их число. Четыре бита требуется для записи десятичной двоично-кодированной цифры. По-видимому, можно утверждать, что лишь небольшая часть памяти используется для записи таких цифр и вряд ли стоит иметь ячейку такой длины. Следующими по длине являются символы. Символы — весьма распространенная единица информации. Это не только данные. Сами программы, написанные на каких-либо языках программирования, вводятся в память в виде символов. Как уже говорилось, первоначально для записи одного символа отводилось 6 бит. Однако по мере расширения использования компьютеров для обработки символьной

информации увеличивалось и число используемых символов. Кроме букв, прописных и строчных, знаков препинания и цифр, которые внутри текста кодируются как символы, стали записывать знаки процентов, градусов, долларов и фунтов стерлингов, параграфов и т. д. Их число быстро перевалило за 64, и стало необходимо использовать комбинации из 7 бит. В конце концов, было признано целесообразным иметь ячейку длиной в 8 бит. Во-первых, было ясно, что число символов может еще увеличиться и целесообразно иметь один бит в запасе, а во-вторых, в 8 бит записываются ровно две десятичные двоично-кодированные цифры.

Такую ячейку назвали *байтом*. Байтом также называют количество информации, которую можно хранить в ячейке из восьми бит.

Байтовая память в принципе позволяет организовывать логические ячейки из произвольного числа байтов. Однако устройство процессора будет проще, если согласиться на некоторые ограничения. Принято для каждого типа компьютера определять основную единицу памяти, ту, которая, как предполагается, будет чаще всего использоваться. Ее называют *словом* (иногда — машинным словом). В большинстве машин слово имеет длину два или четыре байта. Вводятся и другие единицы памяти. Двойное или длинное слово состоит из двух слов. Иногда вводится также сверхдлинное слово — четыре слова. В случае, если слово состоит из четырех байт, вводится полуслово.

Однако мало провозгласить новые единицы памяти, нужно изменить процессор таким образом, чтобы он мог работать с каждой из них. Для этого процессор должен быть способен решать две новые задачи. Первая — каждый раз, вызывая команду, определять ее длину, вторая — перед выполнением команды определять длину операндов.

Для решения первой задачи в команде выделяется головная часть, которая первой считывается в регистр команд. Если в компьютере данного типа бывают команды длиной в один байт, то головная часть — один байт, если длина команды не менее двух байт, то два байта. Головная часть должна содержать информацию о длине всей команды. Руководствуясь ею, процессор считывает оставшуюся часть команды или определяет, что считана уже вся команда.

Информация о длине команды используется также для продвижения УК. Очевидно, что при исполнении команды арифметико-логического типа к содержимому УК нужно прибавлять длину этой команды в байтах.

Длина операндов определяется после считывания команды. Обычно информация о ней также содержится в головной части.

Команды с регистровыми адресами имеют два достоинства. Во-первых, они короткие. Регистров в компьютере немного — 8, 16, 128, редко больше. По-

этому для их адресов нужны небольшие поля — 3, 4, 7 бит на каждый регистр. Во-вторых, они выполняются очень быстро. Особенно команды формата "регистр — регистр". Им требуется всего одно обращение к памяти — для извлечения самой команды, а данные выбираются из регистров и засылаются в регистры. Так как большинство промежуточных данных требуется почти сразу после их вычисления, программист имеет возможность для большинства команд использовать формат "регистр — регистр".

3.4. Косвенные, непосредственные и относительные адреса

Адреса, упоминаемые в команде, различаются по способу указания на память. До сих пор речь шла только о прямых адресах памяти и прямых регистровых адресах. Кроме них используются также косвенный, непосредственный и относительный адреса.

3.4.1. Косвенный адрес

Косвенный адрес — это адрес адреса. Понятно, что ячейка памяти может хранить не только данные, но и адреса (ведь это — целые числа). Пусть x хранится в ячейке A. Тогда A — прямой адрес x. Теперь пусть число A хранится в ячейке B. Тогда B — прямой адрес A, следовательно, B — адрес адреса x. То есть B — косвенный адрес x. Кратко: если $(A)=x$, а $(B)=A$, то $((B))=x$.

На рис. 3.19 изображен участок памяти с ячейками A и B.

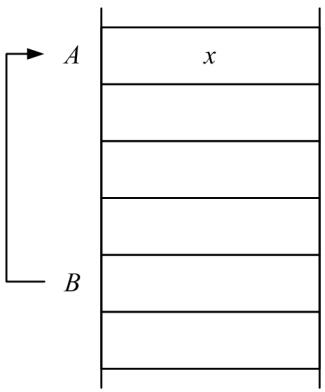


Рис. 3.19. Косвенная адресация

Конечно, немного странно хранить адрес адреса переменной. Но косвенные адреса необходимы, если в последовательных ячейках памяти хранятся какие-то наборы данных — векторы, матрицы и т. д. или фрагменты программы, или файлы. Тогда необходимо хранить адрес первой ячейки соответствующей последовательности, т. е. адрес адреса. Если в компьютере не предусмотрено использование командами косвенных адресов, как и было в машинах первых поколений, то для извлечения элемента последовательности, заданной косвенным адресом, придется выполнить несколько команд.

Чтобы различать прямые и косвенные адреса, где-то в команде, например, в полях адресов, как на рис. 3.20, вводятся признаки. $\mu_i = 0$ означает, что адрес Ai — прямой, $\mu_i = 1$, — что адрес Ai — косвенный.

Команды типа "регистр — регистр" могут использовать оба прямых или оба косвенных адреса. Один из адресов может быть прямым, другой — косвенным. Варианты различаются признаками в команде. Двухадресная команда "регистр — регистр" может иметь, например, вид, как на рис. 3.20.

P	μ_1 $R1$	μ_2 $R2$

Рис. 3.20. Команда с прямыми и косвенными адресами

Такая команда выполняется следующим образом:

1. $((\text{УК})) \rightarrow$ РК.
2. $(\text{УК}) + l \rightarrow$ УК.
3. Дешифрировать код операции.
4. Если $\mu_1 = 0$, то $(R1) \rightarrow r1$, иначе — $((R1)) \rightarrow r1$.
5. Если $\mu_2 = 0$, то $(R2) \rightarrow r2$, иначе — $((R2)) \rightarrow r2$.
6. $(r1)\,p\,(r2) \rightarrow r3$.
7. Если $\mu_2 = 0$, то $(r3) \rightarrow R2$, иначе — $(r3) \rightarrow R2$.
8. Перейти к выполнению пункта 1.

В пункте 2 описания этапов выполнения команды к содержимому указателя команд прибавляется длина команды в байтах — l вместо прежней единицы.

Использование в команде косвенных адресов имеет важный недостаток — каждый косвенный регистровый адрес требует обращения к памяти. Таким образом, команда формата "регистр — регистр" с косвенными адресами требует для выполнения столько же обращений к памяти, как и команда с пря-

мыми адресами, и выполняется даже несколько дольше, т. к. затрачивает дополнительное время на формирование адреса памяти. Но все же, она значительно короче команды с прямыми адресами и вызывается из памяти быстрее.

3.4.2. Непосредственный адрес

Как уже было сказано, память компьютера хранит команды и данные. Рассмотрим этот вопрос более подробно. Представим себе память компьютера непосредственно перед началом выполнения программы. Часть ее занята программой, часть занята исходными данными, часть — свободна (рис. 3.21).

Рис. 3.21

Часть свободных ячеек предназначена для размещения промежуточных и конечных результатов и в процессе исполнения программы будет использована. Исходные данные делятся на переменные и константы. Программа состоит из команд (рис. 3.22).

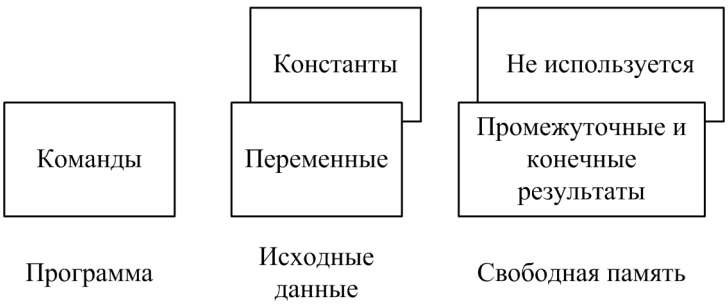


Рис. 3.22

Программа предназначена для многократного использования с различными вариантами исходных данных. На рис. 3.23 представлена схема использования программы.

Пунктирные стрелки показывают, что ввод части исходных данных и вывод части конечных результатов может производиться в процессе вычислений. В любом случае каждый новый вариант вычислений будет связан с новым вводом исходных данных.

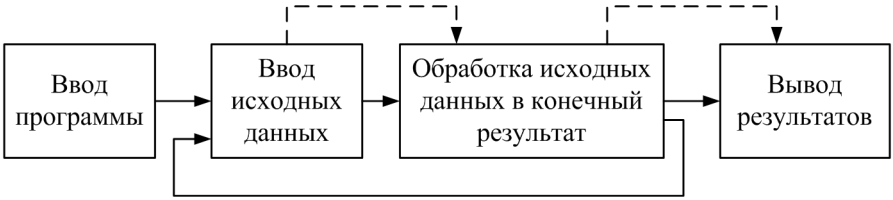


Рис. 3.23

Однако часть данных во всех вариантах будет сохранять свое значение. Это и есть константы. Например, тригонометрическую функцию $y = \sin x$ при малых x можно вычислять по формуле

$$y = x - \frac{x^3}{6}.$$

Здесь x — переменная, 6 — константа.

Константы нецелесообразно вводить по много раз, поэтому их обычно присоединяют к программе и вводят вместе с ней один раз.

Промежуточные и конечные результаты, поскольку они, очевидно, не являются константами, также можно отнести к переменным. Тем более что, как в одном из примеров, рассмотренных ранее, одни и те же ячейки можно использовать для исходных переменных и промежуточных и конечных результатов. В таком случае распределение памяти выглядит как на рис. 3.24.

Рис. 3.24

Константы обычно располагают в памяти сразу после последней команды программы.

Будем делить константы на одноразовые и многократного использования. Одноразовая используется только один раз, в одной команде данной программы. Многоразовые используются различными командами программы.

Логично константы, нужные многим командам, держать в каком-то общем месте, а одноразовые — вблизи того места, где они используются.

В машинах с байтовой организацией памяти такая возможность имеется. Константу помещают прямо за использующей ее командой. Вместе они считаются единой командой. То есть команда имеет вид, как показано на рис. 3.25.

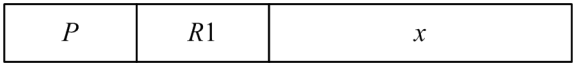


Рис. 3.25

В случае, если $R1$ — прямой адрес, команда исполняется по такой схеме:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $(R1) \rightarrow r1$.
5. $x \rightarrow r2$.
6. $(r1)\,p\,(r2) \rightarrow r3$.
7. $(r3) \rightarrow R1$.
8. Перейти к выполнению пункта 1.

Аналогично выполняется операция с косвенным первым адресом.

Формально считается, что команда всегда состоит из кода операции и адресной части. Поэтому константу x, стоящую на месте второго адреса, так и считают вторым адресом, но, в отличие от прямого или косвенного называют непосредственным адресом.

Часто в командах с прямыми и косвенными адресами имеется признак направления, указывающий направление засылки результата — на место первого операнда или второго.

В командах с непосредственными адресами нет вариантов. Результат всегда засылается в $R1$. Надо иметь в виду, что участок программы, содержащий данную команду, может использоваться многократно с новыми исходными данными и константу нельзя испортить.

3.4.3. Использование регистрового и непосредственного адресов для формирования адресов памяти

Ранее говорилось о машинах с короткими командами. Длина этих команд достаточна лишь для адресации регистров.

В таких машинах возникает следующая проблема: как в регистры могут попасть переменные или их адреса? Ведь ввести извне информацию можно только в оперативную память.

Покажем, как это можно сделать с помощью коротких команд двух форматов: "регистр — регистр" и "регистр — непосредственный операнд", изображенных на рис. 3.26.

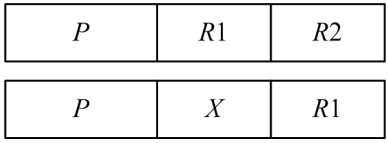


Рис. 3.26

Пусть, например, надо сложить x и y из ячеек с адресами $ABCD$ и $ABCD+2$. Задачу решает такая последовательность команд:

1. L A $R1$ {загрузка непосредственного операнда A в регистр}.
2. SLL 4 $R1$ {сдвиг влево логический на 4 разряда содержимого $R1$}.
3. A B $R1$ {$(B)+(R1)\to R1$; теперь $(R1)=AB$}.
4. SLL 8 $R1$ {сдвиг влево на 8 разрядов $(R1)$; теперь $(R1)=AB00$}.
5. L C $R2$.
6. SLL 4 $R2$.
7. A D $R2$ {в $R2$ сформировано число CD}.
8. $A1$ $R2$ $R1$ {в $R1$ сформировано число $ABCD$ — адрес x}.
9. C $R1$ $R2$ {копирование $(R1)\to R2$}.
10. A 2 $R2$ {в $R2$ сформировано число $ABCD+2$ — адрес y}.
11. $A2$ $R1$ $R2$ {в $ABCD+2$ получена сумма $x+y$}.

Смысл кодов операций ясен из комментариев. Различие в кодах A, $A1$ и $A2$ объясняется тем, что в таких командах по-разному интерпретируется адрес-

ная часть: в первом случае это код сложения непосредственного операнда с содержимым прямого регистрового адреса, во втором — содержимого двух прямых адресов, а в третьем — содержимого косвенных адресов.

4.4.4. Относительный адрес

Выделим еще одну группу констант — константы локального использования. То есть константы многоразового использования разделим на две группы — локальные и глобальные. Локальность константы определяется относительно конкретной команды. Если константа находится вблизи команды не дальше λ байт, то она локальна по отношению к данной команде. Таким образом, одна и та же константа может быть локальной для одной команды и глобальной для другой. Значение λ зависит от системы команд. Это может быть, например, 128 или 256, или даже 2^{16}.

Константа, используемая командой, часто оказывается локальной относительно этой команды. На такие локальные константы и рассчитана относительная адресация.

Двухадресная команда с относительной адресацией короткая, короче, чем команда формата "регистр — память", и имеет вид, представленный на рис. 3.27.

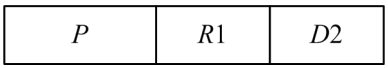


Рис. 3.27

Процессор интерпретирует $D2$ как целое со знаком и использует его в качестве смещения относительно указателя команд. Команда выполняется по схеме:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $(R1) \rightarrow r1$.
5. ((УК) + $D2) \rightarrow r2$.
6. $(r1)\,p\,(r2) \rightarrow r3$.
7. $(r3) \rightarrow R1$.
8. Перейти к выполнению пункта 1.

Подчеркнем еще раз: $D2$ — целое со знаком, так что константа может находиться в памяти как после команды, так и до нее. По адресу (УК) + $D2$ хранится константа, так что направление засылки результата возможно только одно — первый адрес.

Конечно, существуют и другие разновидности этой команды. Во-первых, первый адрес может быть и косвенным регистровым и прямым адресом памяти. Во-вторых, смещение может быть большей длины. В-третьих, иногда смещение умножается на 2, так что второй адрес формируется по правилу (УК) + $2 \times D2$.

3.5. Пересылки

Для объяснения операций пересылок необходимо рассмотреть более сложную схему процессора и его взаимодействия с оперативной памятью (рис. 3.28).

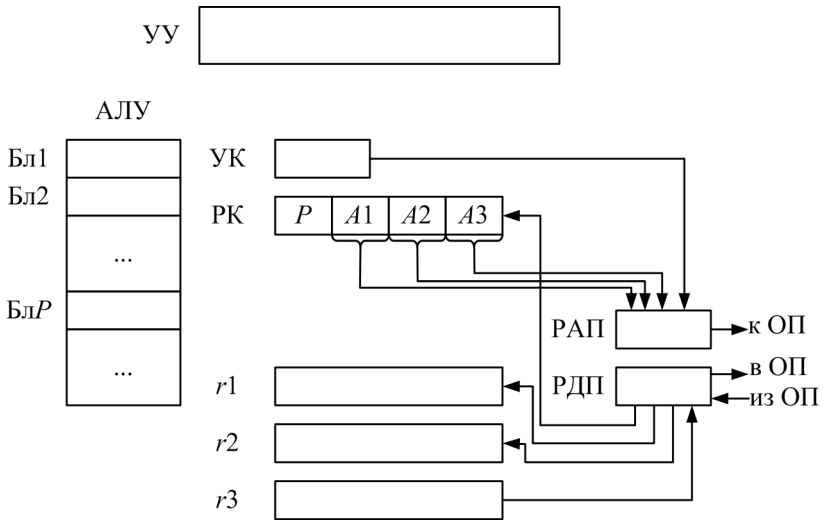


Рис. 3.28. Схема взаимодействия процессора с памятью

На схеме появились два новых элемента: РАП — регистр адреса памяти и РДП — регистр данных памяти.

Они заменили изображавшуюся прежде память. Через эти два регистра по шинам процессор связывается с оперативной памятью. По адресной шине в память передается адрес, а по шине данных из памяти или в память передаются данные. Каждый раз, когда процессор обращается к памяти — для чте-

ния команды или данных или для записи результата — он засылает адрес памяти в РАП (часто говорят: "выставляет адрес"), а по управляющей шине сигнализирует памяти, требуется ли прочитать содержимое этого адреса или записать по нему новое значение. По команде "читать" память пересылает содержимое выставленного адреса в РДП. Если память получила команду "записать", то содержимое РДП пересылается в память адресу в РАП. Конечно, процессор к этому времени должен заслать нужное значение в РДП. Стрелки на схеме указывают направления пересылки информации.

Основные этапы работы процессора при исполнении команды арифметического типа, описанные ранее, можно уточнить таким образом:

1. (УК) $\rightarrow$ РАП, выдать управляющий сигнал "читать" {(РАП) $\rightarrow$ РДП, (РДП) $\rightarrow$ УК}.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $A1 \rightarrow$ РАП, выдать управляющий сигнал "читать" {(РАП) $\rightarrow$ РДП, (РДП) $\rightarrow r1$}.
5. $A2 \rightarrow$ РАП, выдать управляющий сигнал "читать" {(РАП) $\rightarrow$ РДП, (РДП) $\rightarrow r2$}.
6. $(r1)p(r2) \rightarrow r3$.
7. $(r3) \rightarrow$ РДП; $A3 \rightarrow$ РАП; выдать управляющий сигнал "записать" {(РДП) $\rightarrow$ РАП}.
8. Перейти к выполнению пункта 1.

В пунктах 1, 4, 5 и 7 действия, заключенные в фигурные скобки, выполняются не процессором, а памятью по указаниям процессора. Точнее, специальной электронной схемой, называемой контроллером памяти.

В компьютере любого типа имеется группа команд, называемых *пересылками* или *командами обмена*. Сюда относятся команды формата "регистр — регистр", "регистр — память", "память — регистр" и "память — память", создающие копии данных: $(R1) \rightarrow R2$, $(R1) \rightarrow A2$, $(A1) \rightarrow R2$, $(A1) \rightarrow A2$.

Они особенно распространены в двухадресных и одноадресных машинах, где создание копии необходимо для сохранения данных, уничтожаемых арифметико-логическими операциями. Кроме того, операции с регистрами выполняются быстрее, поэтому целесообразно загружать в регистры наиболее используемые в настоящий момент данные и выгружать данные, которые в ближайшее время не понадобятся.

Все эти операции — двухадресные. Данное передается из источника в приемник. Команды пересылки между регистрами, между ячейками памяти и

между регистрами и памятью выполняются по той же схеме, что и уже рассмотренные команды арифметико-логического типа. Тут нечего добавить. Важно лишь подчеркнуть, что все пересылки выполняются через процессор, точнее, через его РДП. То есть операция пересылки содержимого ячейки A в ячейку B — $(A) \to B$ — выполняется (опуская подробности) в два шага:

1. $(A) \to$ РДП.
2. (РДП) $\to B$.

То есть пересылки выполняются с участием процессора.

3.5.1. Обмен с внешней памятью

К группе пересылок можно отнести и команды обмена между внешней памятью и основной, а также команды ввода и вывода. Их обычно называют просто "операциями обмена", а также "операциями ввода/вывода".

Эти операции должны "по идее" также выполняться через процессор. Только так они и выполнялись первоначально. Затем, дополнительно к этому, появилась и другая возможность — прямой доступ к памяти. Варианты доступа к памяти будут рассмотрены позже.

Внешние устройства непосредственно управляются контроллерами — специализированными процессорами. В составе контроллера имеется небольшой набор собственных регистров, в частности регистр данных и регистр состояния. Регистр данных контроллера играет такую же роль, как и регистр данных памяти (РДП) — роль окна, через которое производится передача данных.

Регистр состояния хранит информацию о готовности внешнего устройства к работе. В контроллере имеется также регистр вида работ — ввод или вывод (IN/OUT).

Для взаимодействия программы с внешним устройством обычно требуется больше одной команды. Для обмена программа должна задать: адрес ячейки оперативной памяти, из которой (или в которую) надо производить пересылку, адрес внешнего устройства, с которым надо произвести обмен, направление обмена (ввод или вывод), номер регистра общего назначения, в который надо прочитать информацию о состоянии устройства. Для некоторых устройств надо сообщить также дополнительную информацию. Если, например, обмен производится с диском, надо указать также адрес на диске.

Как и для регистров общего назначения, возможны два варианта адресации внешних устройств.

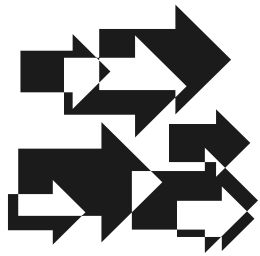
Первый — с использованием общего адресного пространства (иначе — "с отображением внешних устройств в память"). В этом случае регистры кон-

троллеров заменяют некоторые ячейки памяти. Если, например, регистр данных контроллера некоторого устройства имеет номер 100, то из числа ячеек ОП исключается ячейка 100 и ее заменяет регистр данных контроллера устройства 100. И если, например, встретится команда "сложить содержимое регистра *R*1 с содержимым ячейки 100", то в операции будет участвовать регистр данных контроллера устройства 100. В этом случае не требуется специальных команд обращения к внешним устройствам.

Второй вариант — с использованием отдельного адресного пространства для внешних устройств. В этом случае имеются одинаковые номера у регистров контроллеров и у некоторых ячеек. Чтобы их различать, вводятся специальные команды, которые обращаются к регистрам контроллера — команды ввода и вывода. Все остальные команды могут обращаться только к ячейкам памяти. Таким образом, в команде "сложить содержимое регистра *R*1 с содержимым ячейки 100" будет участвовать ячейка памяти 100, а в команде "ввести в регистр *R*1 данное из устройства 100" будет участвовать регистр данных устройства 100.

В современных компьютерах чаще используется вариант с отдельным адресным пространством. В этом случае все подготовительные операции для обмена являются пересылками между памятью (или регистрами общего назначения) и регистрами внешних устройств.

ГЛАВА 4



Команды передачи управления и циклы

4.1. Переходы

4.1.1. Разветвления в алгоритмах и программах

Важный элемент алгоритмов — разветвление. Это такой момент в процессе переработки информации, когда возможны альтернативные варианты продолжения. Проверяется некоторое условие, и если оно выполнено, то процесс продолжается по одному сценарию, если нет — то по-другому. Блок-схема части алгоритма, прилегающей к разветвлению, изображена на рис. 4.1.

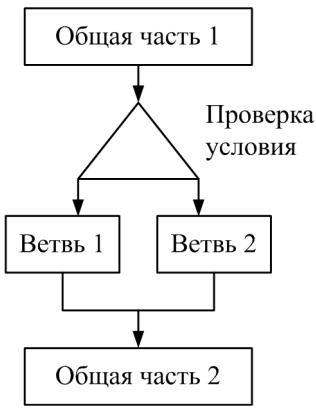


Рис. 4.1. Блок-схема разветвления

Разветвления в программах организуются с помощью команд передачи управления (их также называют командами перехода, командами ветвления).

Различают команды условного перехода и безусловного.

Реализация в виде программы приведенной на рис. 4.1 схемы требует двух команд перехода: одной — условного и одной — безусловного (соответствующий фрагмент программы — на рис. 4.2). Это связано с тем, что память компьютера линейна и ветви алгоритма располагаются в ней последовательно, а не параллельно, как на схеме.

После общей части 1 проверяется некоторое условие. Если условие не выполнено, то дальнейшие вычисления ведутся по ветви 1, если выполнено, — по ветви 2. После окончания любой из них идет общее продолжение — общая часть 2.

Отметим, что весьма распространенным является частный случай разветвления, когда ветвь 2 отсутствует, как на рис. 4.3.

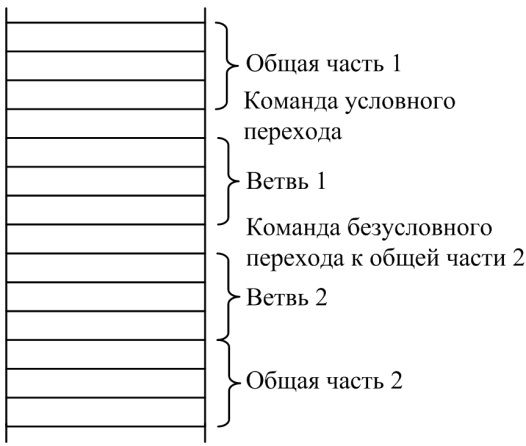


Рис. 4.2. Последовательное расположение ветвей алгоритма в памяти

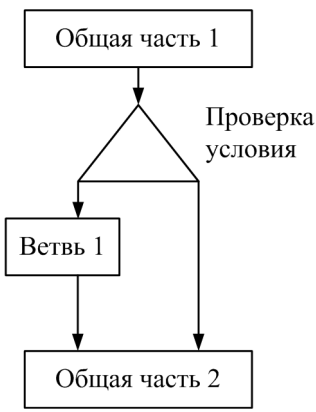


Рис. 4.3. Разветвление при отсутствии ветви 2

Тогда команда безусловного перехода не нужна. Команда условного перехода определяет, выполнять ветвь 1 или пропустить. Соответствующий фрагмент программы приведен на рис. 4.4.

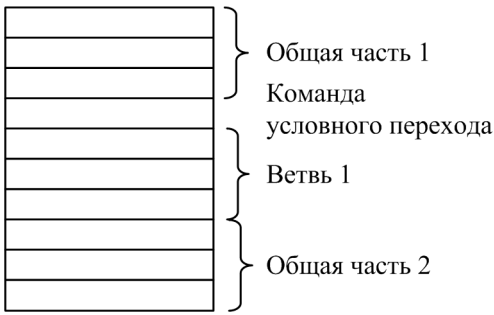


Рис. 4.4. Фрагмент программы с разветвлением при отсутствии ветви 2

4.1.2. Безусловные переходы

Команда безусловного перехода — простейшая из команд передачи управления. Эта команда состоит из кода операции и адреса перехода. То есть формально она может быть двухадресной или трехадресной, но в любом случае используется только один из адресов.

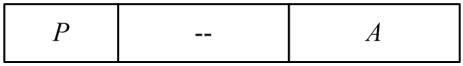


Рис. 4.5

Основные этапы выполнения такой команды следующие:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК {l — длина команды в байтах}.
3. Дешифрировать код операции.
4. A $\rightarrow$ УК.
5. Перейти к выполнению пункта 1.

Здесь и далее опускаются подробности, описывающие использование регистров РАП и РДП.

Может использоваться регистровая адресация, т. е. команда вида, представленного на рис. 4.6.

Тогда пункт 4 выполняется иначе — (R) $\rightarrow$ УК.

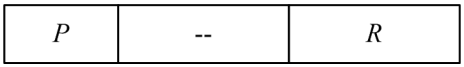


Рис. 4.6

В первом случае в команде используется непосредственный адрес, а во втором — прямой регистровый. В машинах, использующих команды переменной длины, последняя команда имеет вид, показанный на рис. 4.7.

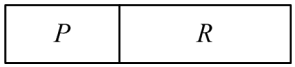


Рис. 4.7

Она короче, а исполняется за то же время, что и команда с прямым адресом. Однако надо иметь в виду, что адрес перехода в R надо занести, а это — дополнительная команда.

4.1.3. Условные переходы. Признаки результата

Большинство команд передачи управления — условные переходы.

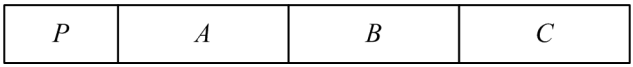


Рис. 4.8. Трехадресная команда условного перехода

В трехадресных машинах команды условного перехода могут выполняться, в зависимости от кода операции, по одной из схем:

1. Если $(A) < (B)$, то $C \to$ УК.
2. Если $(A) > (B)$, то $C \to$ УК.
3. Если $(A) = (B)$, то $C \to$ УК.
4. Если $(A) \neq (B)$, то $C \to$ УК.

Могут быть также команды, с комбинацией условий: "если $(A) \leq (B)$, то $C \to$ УК " и "если $(A) \geq (B)$, то $C \to$ УК ". Во всех случаях при невыполнении условия происходит переход к следующей команде.

В двухадресных и одноадресных машинах (да и в большинстве современных трехадресных) организация перехода разделена на две стадии: подготовка условия и проверка. Проверку условия и переход, если условие выполнено,

по-прежнему осуществляет команда передачи управления, а формирование условия — предыдущая команда.

Для передачи значения истинности условия в процессоре существует еще один регистр — регистр флагов (РФ). Обычно часть разрядов РФ не используется (зарезервировано), а остальные делятся на управляющие, системные и разряды состояния. Нас будут интересовать только последние. Иногда совокупность разрядов состояния условно называют регистром признаков результата — РПр.

Многие команды арифметико-логического типа, кроме основного результата, помещаемого в *r*3, вырабатывают еще и признак результата — бит, который записывается в один из разрядов состояния РФ. Содержимое этих разрядов называют также флагами. Говорят, например: "в разряде три установлен флаг", если туда записана единица, а если ноль, то — "флаг сброшен". Обычно флагов состояния 4—6. Обязательно имеются флаги нуля — *ZF* (Zero Flag), знака — *SF* (Sign Flag), переполнения — *OF* (Overflow Flag). Флаг SF устанавливается, если результат отрицателен, *ZF* — если результат — ноль; *OF* — при переполнении. В большинстве случаев для управления алгоритмами используются вообще только первые два флага.

Для двухадресной машины основные такты выполнения команды арифметического типа с учетом регистра флагов выглядят так:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $(A1) \rightarrow r1$.
5. $(A1) \rightarrow r2$.
6. $(r1)\,p(r2) \rightarrow r3$.
7. Признак результата заслать в РФ.
8. $(r3) \rightarrow A2$.
9. Перейти к выполнению пункта 1.

Те команды, которые не вырабатывают признака результата, сохраняют прежнее значение в РФ.

Каждой команде условного перехода (рис. 4.9) с кодом операции P ставится в соответствие определенное число N_p — набор флагов регистра признаков результата, которые влияют на выполнение данной команды.

Большинство команд условных переходов можно объединить в пары — антиподы: одна передает управление при совпадении хотя бы одного выставленного флага в РП и в наборе N_p, другая, — если не совпадает ни один.

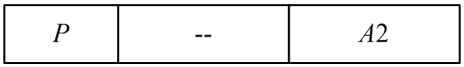


Рис. 4.9

Пары команд такого вида выполняются по схеме:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. Если $\left[\left(\text{РПр}\right)\text{AND}\,(N_p)\right] \neq 0$, то $A2 \rightarrow$ УК.
5. Перейти к выполнению пункта 1.

Или по противоположной:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. Если $\left[\left(\text{РПр}\right)\text{AND}\,(N_p)\right] = 0$, то $A2 \rightarrow$ УК.
5. Перейти к выполнению пункта 1.

Так же выполняются команды с регистровой адресацией (рис. 4.10).

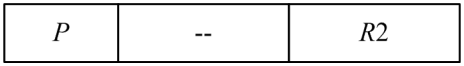


Рис. 4.10

Разница только в одном пункте:

4. Если $\left[(\text{РПр})\,\text{AND}\,(N_p)\right] \neq 0$, то $R2 \rightarrow$ УК.

Или

4. Если $\left[\left(\text{РПр}\right)\text{AND}\,(N_p)\right] = 0$, то $R2 \rightarrow$ УК.

Необходимо уточнить процесс формирования условия перехода. На команду условной передачи управления непосредственно влияет лишь та из предшествующих ей команд, которая последней выработала признак результата. Однако эта команда может только завершать процедуру вычисления условия. Часто вся процедура состоит из одной команды, но может насчитывать и несколько команд. То есть из "общей части 1" на рис. 4.2 можно выделить блок

подготовки проверки условия перехода. Этот блок не принимает непосредственного участия в получении конечного результата, а нужен лишь для управления процессом вычислений, для получения признака. В связи с этим конечной командой такого блока обычно является команда сравнения, которая выполняется как вычитание, но без записи результата в память. При этом признак результата в РПр записывается.

4.1.4. Безусловные и условные переходы по смещению

В машинах, использующих команды переменной длины, команды переходов более короткие, т. к. в них исключается неиспользуемый адрес (рис. 4.11).

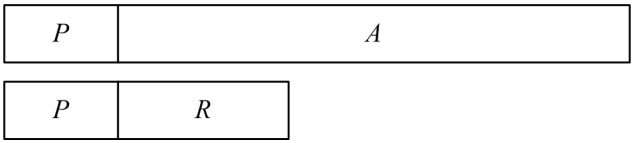


Рис. 4.11

Поле адреса в первой из них большое — оно должно вмещать самый большой номер ячейки. Даже в первых персональных компьютерах для этого были нужны два байта, а в современных — все четыре. Поэтому команда с регистровой адресацией предпочтительнее с точки зрения экономии памяти: в ней для поля адреса вполне достаточно одного байта.

Однако и у нее есть недостатки. Главный — чтобы передать управление по адресу, хранящемуся в регистре, надо, чтобы в регистре этот адрес был, а значит, где-то раньше должна быть выполнена дополнительная команда — засылки адреса в регистр.

В связи с этим для команд перехода используется еще один способ адресации — *относительная адресация*. Команды с относительной адресацией могут иметь вид команд формата "регистр — регистр", но в адресной части вместо номеров регистров записывается целое число со знаком смещение D (рис. 4.12).

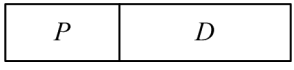


Рис. 4.12

В отличие от других команд переходов здесь не засылается в УК новое значение, а число D прибавляется к УК.

Вот алгоритм выполнения этой команды:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + l $\rightarrow$ УК.
3. Дешифрировать код операции.
4. Если $\left[\left(\text{РПр}\right)\text{ AND }(N_p)\right] \neq 0$, то $(\text{УК}) + D \rightarrow \text{УК}$.

   Или:

   Если $\left[\left(\text{РПр}\right)\text{ AND } N_p\right] = 0$, то $(\text{УК}) + D \rightarrow \text{УК}$.
5. Перейти к выполнению пункта 1.

Важно помнить, что при исполнении команды сначала продвигается УК (прибавляется l) и лишь затем прибавляется смещение. Рисунок 4.13 иллюстрирует это на примере смещений –3 и +3. Заметьте, смещение симметрично не относительно команды, передающей управление, а относительно следующей.

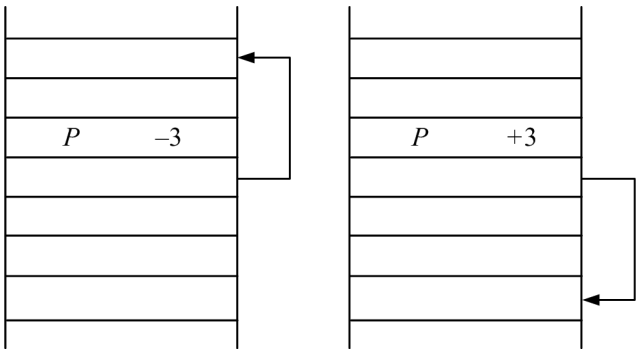


Рис. 4.13

Надо сказать, что и эта картинка достаточно условна и служит лишь для иллюстрации принципа. Она верна, если все команды, прилегающие к разветвлению, имеют длину в один байт.

Пусть команда передачи управления имеет длину l_1 байт, а следующая за ней — l_2. Тогда схемы передачи управления по положительному или отрицательному смещению D имеют вид, как на рис. 4.14.

Команда передачи по смещению такая же короткая, как и регистровая, но не требует предварительной засылки адреса перехода. Правда, и у нее есть свой недостаток — она не может передавать управление далеко — всего на

±127 байт, если для записи смещения выделен один байт. Иногда смещение производится на $2D$, но и это не далеко. Однако в программах редко требуются далекие передачи управления, поэтому такие команды употребляются часто.

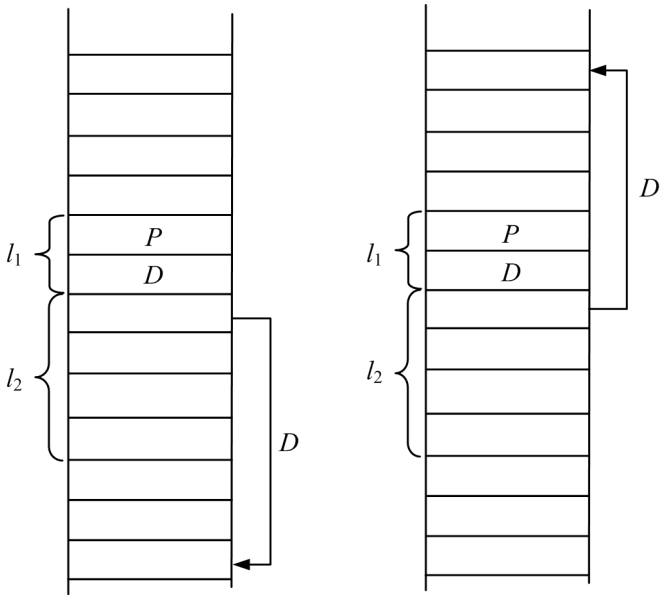


Рис. 4.14

Но и далекие передачи управления, хотя и редко, но нужны. В то же время, чем больше различных вариантов команд, тем дороже процессор. Поэтому в системах команд обычно используется компромиссное решение: имеются как близкие, так и далекие команды безусловной передачи управления и только близкие (по смещению) условные передачи управления. А так как команд условной передачи управления на порядок больше, чем безусловной, то экономия на количестве различных команд получается значительной.

А проблема условного далекого перехода решается с помощью приема, схематически изображенного на рис. 4.15. Слева — организация перехода с помощью одной команды условного перехода (возможен только близкий переход), справа — переход в ту же точку программы с помощью комбинации команд условного близкого перехода (антипода команды, использованной слева) и безусловного далекого (возможен как близкий, так и далекий переход).

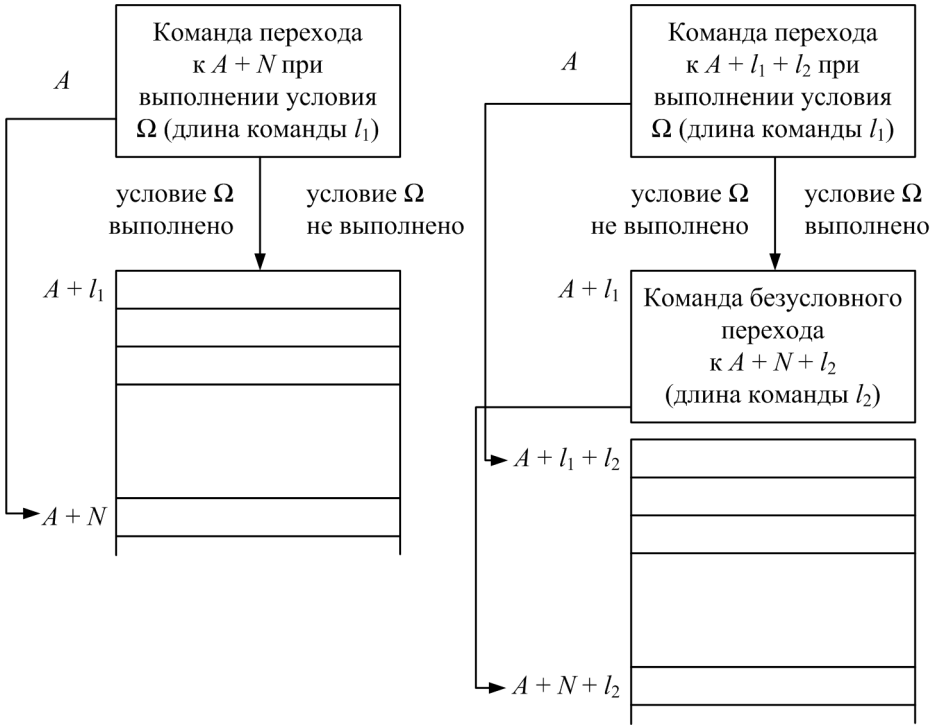


Рис. 4.15. Далекий условный переход

4.2. Циклы

4.2.1. Классификация циклов

Цикл — участок программы, обеспечивающий многократное выполнение некоторой части алгоритма до тех пор, пока не выполнится условие окончания. Цикл состоит из внутренней части, выполняющейся многократно, и внешней — команд подготовки и команд завершения цикла, выполняющихся по одному разу. Многократное выполнение обеспечивается тем, что во внутренней части цикла обязательно присутствует команда условной передачи управления.

Выделим последовательность ячеек, содержащую цикл (рис. 4.16).

Первая команда внутренней части цикла — та, на которую указывает стрелка, последняя — та, откуда стрелка выходит. Последняя команда всегда является командой передачи управления. Команды, заключенные между первой и последней, составляют тело цикла. К телу цикла могут сверху примыкать команды, его подготавливающие, снизу — его завершающие.

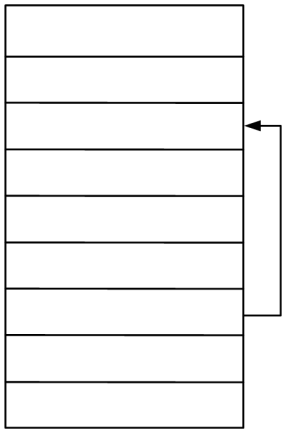


Рис. 4.16

В теле цикла должна иметься хотя бы одна команда условного перехода. Она определяет, нужно ли выполнить команды, входящие в тело цикла, еще раз или закончить цикл. Эту команду можно расположить в конце тела цикла или в его начале. Принципиальной разницы между этими вариантами нет. Обычно первая схема принимается в том случае, если ясно, что число выполнений цикла, во всяком случае, не меньше одного, вторая, — если существует возможность, что тело цикла не нужно будет выполнять ни одного раза. В этом случае в конце тела цикла должна находиться команда безусловной передачи управления на его начало. Как при любом разветвлении в теле цикла команде условной передачи управления предшествует блок подготовки перехода.

На рис. 4.17 представлены схемы цикла для случая расположения команды условной передачи управления в конце тела цикла (на рисунке слева) и в начале.

Если цикл выполняется по схеме 1 (слева), то основные вычисления и вспомогательные — команды блока подготовки перехода могут быть перемешаны, и лишь последняя команда вспомогательных вычислений должна быть предпоследней в теле цикла, т. к. признак результата именно этой команды должен использоваться командой условного перехода.

Итак, первый способ классификации циклов — по расположению проверки условия перехода: с предшествующей проверкой и с последующей.

Второй способ классификации — по типу алгоритма. Здесь различают три типа циклов:

- ❒ цикл с заданным числом повторений;
- ❒ цикл итерационного типа;
- ❒ цикл смешанного типа.

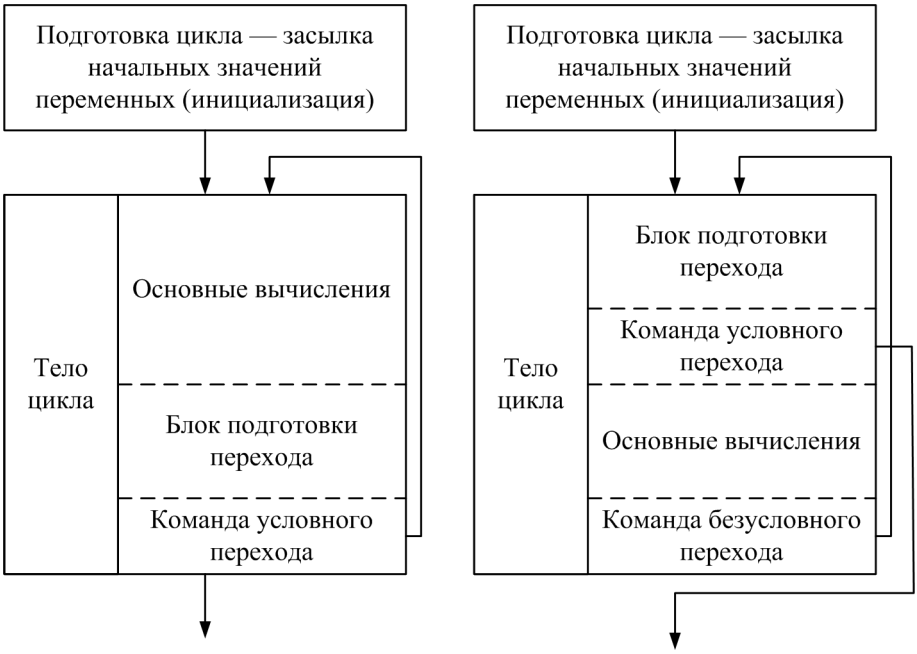


Рис. 4.17. Схемы цикла с проверкой в конце и в начале цикла

Цикл с заданным числом повторений

Такие циклы называют также циклами со счетчиком. Управление ими осуществляется с помощью специально создаваемой переменной, называемой счетчиком.

Задачи, требующие построения цикла со счетчиком, весьма распространены. Это задачи, связанные с различными последовательностями (просуммировать элементы, помножить все элементы на число, найти наибольший элемент и т. д.).

Рассмотрим пример. Пусть требуется вычислить сумму десяти слагаемых вида:

$$\sqrt{1}+\sqrt{2}+\sqrt{3}+...+\sqrt{10}\ .$$

Тогда схема цикла со счетчиком выглядит так, как представлено на рис. 4.18.

Часто ведется обратный отсчет: в счетчике устанавливается нужное число повторений цикла и при каждом прохождении тела цикла из счетчика вычитается 1. Повторение ведется до тех пор, пока в счетчике не установится ноль.

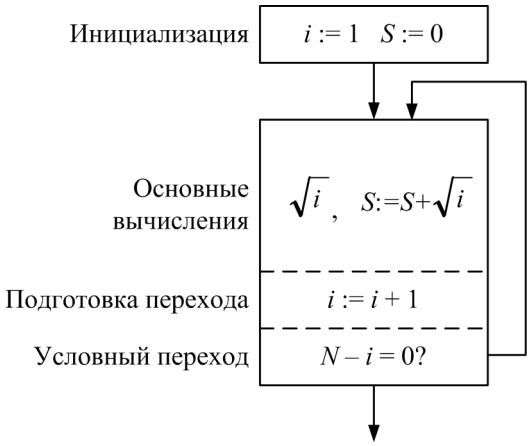


Рис. 4.18. Схема цикла со счетчиком

Цикл итерационного типа

Название этого типа циклов связано с широко распространенными в математике методами последовательных приближений. Другое их название — методы итераций. Эти методы по имеющемуся приближенному решению строят другое решение — более точное. Процесс повторяется до тех пор, пока не будет получено решение с удовлетворительной точностью. Количество повторений заранее не известно.

Рассмотрим пример. Функцию $y = \sin x$ можно приближенно вычислять как частичную сумму ряда

$$y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

При $x < 1$ ряд быстро сходится, а погрешность частичной суммы любого знакопеременного ряда не превышает первого отброшенного члена. Для вычисления y с погрешностью, не превышающей ε, используем такой алгоритм.

Каждое следующее приближение будем получать из предыдущего, прибавляя к нему новое слагаемое, которое, в свою очередь, будем получать из предыдущего:

$$S_1 = x, \quad a_1 = x, \quad S_{i+1} = S_i + a_{i+1}, \quad a_{i+1} = -\frac{a_i x^2}{2i(2i+1)}, \quad i = 1, 2, \ldots$$

Вычисления будем вести до тех пор, пока $|a_{i+1}| - \varepsilon > 0$.

Блок подготовки перехода состоит из двух команд — $|a_{i+1}|$ и $|a_{i+1}| - \varepsilon = ?$. Последняя вырабатывает признак результата для команды условного перехода без засылки результата в память. Перед телом цикла производится его подготовка — в рабочие ячейки (регистры) засылаются начальные значения $i = 1$, $a_i = x$, и $S_i = x$ (рис. 4.19).

Рис. 4.19. Цикл с блоком инициализации

Подчеркнем, итерационный цикл — это не только тот, который реализует методы последовательных приближений, а любой, в котором заранее неизвестно число повторений. Например, когда мы последовательно перебираем список в поисках элемента с определенными признаками, скажем, ищем первое четное число, мы не знаем, сколько элементов списка придется перебрать. Но эту процедуру никто не назовет методом последовательных приближений.

И наоборот, наличие счетчика не делает цикл циклом с заданным числом повторений. В последнем примере переменную i можно назвать счетчиком: i равно числу итераций, однако решающим является то, что счетчик здесь не управляет циклом.

Цикл смешанного типа

Такой цикл имеет два выхода. Пример — вычисление функции $y = \sin x$ с помощью того же ряда, но теперь до тех пор пока не выполнится одно из условий (рис. 4.20):

$$"|a_{i+1}| - \varepsilon > 0" \text{ или } "N - i \neq 0".$$

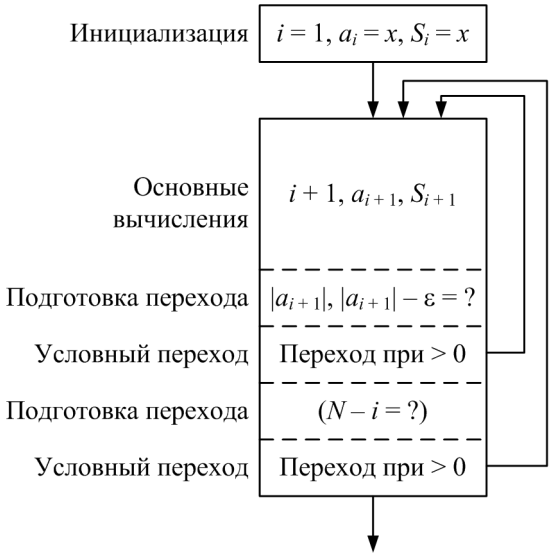


Рис. 4.20. Цикл смешанного типа

Циклы смешанного типа применяются очень широко, по-видимому, чаще, чем итерационного типа. Их можно считать циклами итерационного типа с ограничением числа итераций. Вспомним пример с нахождением нужного элемента в списке. А может, подходящего элемента в списке нет? А мы будем искать без конца. Список давно кончился, но алгоритм об этом не знает и продолжает перебирать ячейку за ячейкой. Чтобы этого не случилось, надо контролировать число проверенных элементов и вовремя закончить поиск.

Кратный цикл

Третий способ классификации циклов — по кратности. Цикл может быть простым, двойным, тройным и т. д. Используются термины "кратный цикл" и "вложенный цикл".

Пусть требуется вычислить значения функции $y = \sin x$ для аргументов $x = x_0,\ x_0/2,\ x_0/3,\ ...,\ x_0/10$.

Соответствующий алгоритм можно записать в виде:

1. Подготовка цикла — засылка начальных значений переменных: $j = 0$.
2. Основные вычисления: $j = j + 1,\ x_j = x_0/j,\ y_j = \sin x_j$.
3. Вспомогательные вычисления: $\delta_j = 10 - j$.
4. Команда условного перехода: "если $\delta_j > 0$, то перейти к пункту 2".

Здесь в тело цикла входит вычисление $y_j = \sin x_j$, которое само реализуется с помощью цикла. Внутренний цикл — итерационного типа, внешний — с заданным числом повторений (рис. 4.21).

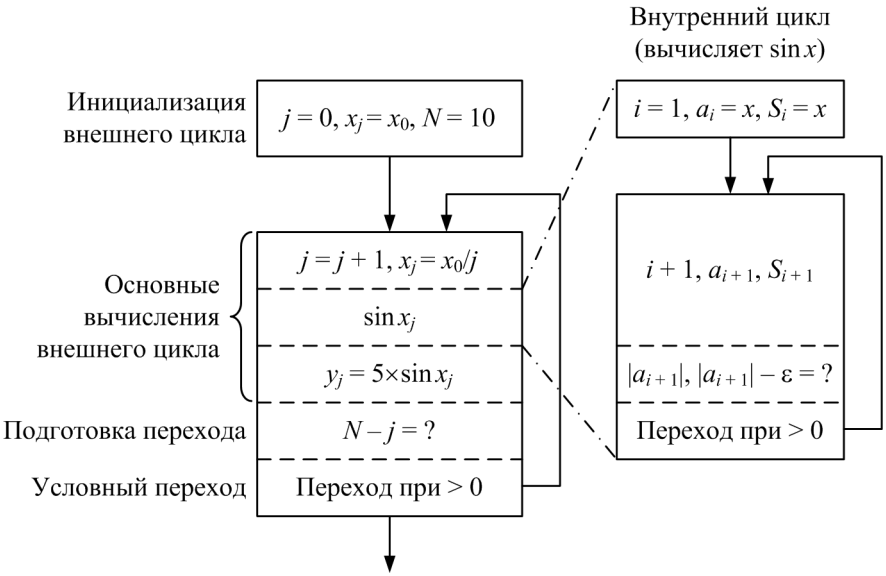


Рис. 4.21. Схема двойного цикла

Отметим важное обстоятельство: цикл можно считать отдельным блоком программы, решающим некоторую частную, логически законченную задачу. Это дает возможность, рассматривая программу в целом, отвлекаться от его внутренней структуры и считать цикл таким же элементом программы, как и команды, разве что более крупным.

4.1.2. Переадресация

Переадресация с помощью констант, восстановление

В следующем примере переменная i не входит в формулы, но все равно ее начальное значение задается в блоке подготовки и текущее значение вычисляется в теле цикла, теперь уже исключительно в качестве счетчика. Здесь ведется обратный отсчет: начальное значение счетчика равно N — числу повторений. С каждым шагом оно уменьшается, пока не станет нулем.

Пусть переменные X_1, X_2, ..., X_n хранятся в последовательных ячейках памяти: $X_1 = (A+1)$, $X_2 = (A+2)$, ..., $X_n = (A+N)$. Пусть $\gamma = (C_\gamma)$.

Пусть требуется вычислить $Y_1 = \gamma X_1$, $Y_2 = \gamma X_2$, …, $Y_n = \gamma X_n$ и разместить результаты в последовательности ячеек — $B+1$, $B+2$, ..., $B+N$.

В трехадресной машине это можно сделать с помощью такого алгоритма:

$(A+1)\times(C_\gamma) \to B+1$

$(A+2)\times(C_\gamma) \to B+2$

...

$(A+N)\times(C_\gamma) \to B+N$

Если N велико (например, больше 10), то явно целесообразнее организовать процесс в виде цикла, в теле которого производятся вычисления по формуле $Y_i = \gamma X_i$. Но для этого надо использовать команду с переменными адресами.

Будем считать, что $(C_1) = 1$, $(C_N) = N$. Пусть D, $D+1$ и т. д. — ячейки, в которых располагаются команды цикла.

1. D $(C_N) \to H$. {В ячейке H — счетчик числа повторений. Команда определяет начальное значение счетчика.}
2. $D+1$ $(A+1)\times(C_\gamma) \to B+1$. {Команда при первом прохождении цикла вычисляет Y_1. В дальнейшем она же должна вычислять Y_2 и т. д.}
3. $D+2$ $(H)-(C_1) \to H$. {Продвигается счетчик.}

Очевидно, цикл должен завершиться командой, проверяющей значение счетчика и заставляющей команды из ячеек $D+1$ и $D+2$ выполняться снова и снова, пока счетчик не сравняется с 0. Однако каждый раз, прежде чем повторять выполнение тела цикла, необходимо продвигать адреса в команде, записанной в ячейке $D+1$. Команда должна сначала принять вид $(A+2)\times(C_\gamma) \to B+2$, затем — $(A+3)\times(C_\gamma) \to B+3$ и т. д.

Изменение адресов команды называется переадресацией. Переадресацию можно выполнить прибавлением (вычитанием) к коду команды специально подобранной константы.

Пусть, например, команда в ячейке $D+1$ имеет вид, как на рис. 4.22 (k и m — число бит в соответствующих полях).

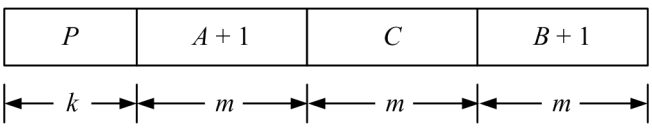


Рис. 4.22

Тогда надо использовать беззнаковую константу, изображенную на рис. 4.23.

00.....0	00.....01	00.....00	00.....01

Рис. 4.23

Пусть она хранится в ячейке C_{const}, а символ $\oplus$ означает беззнаковое сложение. Тогда алгоритм будет иметь вид:

1. $D\ \ (C_1) \to H$ {инициализация счетчика}.
2. $D+1\ \ (A+1)\times(C\gamma) \to B+1$.
3. $D+2\ \ (D+1)\oplus(C_{\text{const}}) \to D+1$.
4. $D+3\ \ (H)-(C_1) \to H$.
5. $D+4\ \ ZF \neq 1$? Да — $\Rightarrow D+1$, нет — $\Rightarrow D+5$.

Последняя команда проверяет значение флага нуля. Пока оно не равно единице (результат предыдущей операции не ноль), цикл повторяется.

Между прочим, в этом алгоритме используется одно из достоинств машин фон-неймановского типа — команда перерабатывается как данное.

Обычно на этом цикл не заканчивается. Представим себе, что этот цикл — внутренний в двойном цикле. Это значит, что, скорее всего, через некоторое время данный цикл будет выполняться снова, например, после того как изменится переменная в ячейке C_γ. Но он не сможет правильно работать, т. к. команда в ячейке $D+1$ изменена посредством переадресации. Очевидно, сразу после выхода из цикла надо восстановить начальный вид команды. Это можно сделать, например, так:

$$D+5\ \ (C_{\text{нач}}) \to D+1,$$

если предварительно запасти в качестве константы "$(A+1)\times(C\gamma) \to B+1$" в ячейке $C_{\text{нач}}$.

Если потребуется вычислять $X_i^2 + Y_i^2 + Z_i^2$, то потребуются две команды переадресации. Легко придумать примеры, где их может быть и больше. Столько же потребуется команд восстановления и констант.

Таким образом, общая структура цикла, с учетом возможной переадресации и восстановления, выглядит так, как показано на рис. 4.24.

Блок восстановления, несмотря на наличие переадресации, может отсутствовать. Вместо восстановления можно в блоке подготовки цикла засылать начальный вид команд в тело программы.

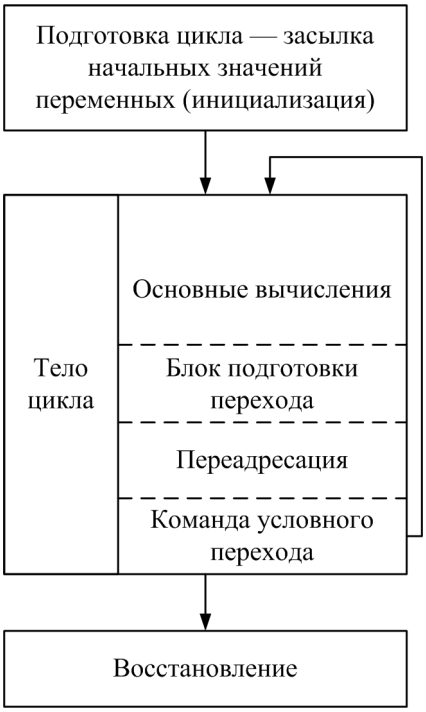


Рис. 4.24. Общая структура цикла

Опыт, накопленный программистами, показал, что на свободу, предоставляемую фон-неймановской архитектурой, полезно наложить ограничения. Команды программы можно перерабатывать как данные, однако лучше, по возможности, этого избегать. В таком случае в программе делается меньше ошибок и облегчается процесс их локализации (попытка переработать команду является формальным признаком ошибки).

Имеются две возможности избежать переменных команд — с помощью косвенных адресов и с помощью индексных регистров.

Косвенные адреса

В машинах с регистрами общего назначения удобно в циклах использовать команды с косвенной регистровой адресацией. Рассмотрим тот же пример с применением регистровых адресов.

1. $D\ \ (C_N) \to H$ {начальное значение счетчика}.
2. $D+1\ \ A+1 \to R1$ {установка указателя на первый элемент массива аргументов}.

3. $D+2$ $B+1 \to R2$ {установка указателя на первый элемент массива результатов}.
4. $D+3$ $((R1)) \times (C\gamma) \to (R2)$ {основные вычисления}.
5. $D+4$ $(R1)+(C_1) \to R1$ {продвижение указателя текущего элемента исходного массива}.
6. $D+5$ $(R2)+(C_1) \to R2$ {продвижение указателя текущего элемента массива результатов}.
7. $D+6$ $(H)-(C_1) \to H$ {продвижение счетчика — выработка признака результата}.
8. $D+7$ $ZF=0$? Да — $\Rightarrow D+8$, нет — $\Rightarrow D+3$.

Однако в действительности, раз уж мы используем машину с РОН, имеет смысл при подготовке цикла перенести используемые в теле цикла константы и переменные в регистры: команд будет больше, а исполняться программа — быстрее.

1. D $(C_N) \to R3$ {начальное значение счетчика}.
2. $D+1$ $A+1 \to R1$ {установка указателя на первый элемент массива аргументов}.
3. $D+2$ $B+1 \to R2$ {установка указателя на первый элемент массива результатов}.
4. $D+3$ $(C_\gamma) \to R4$ $\{\gamma \to R4\}$.
5. $D+4$ $(C_1) \to R5$ $\{1 \to R5\}$.
6. $D+5$ $((R1)) \times (R4) \to (R2)$ {основные вычисления}.
7. $D+6$ $(R1)+(R5) \to R1$ {продвижение указателя текущего элемента массива аргументов}.
8. $D+7$ $(R2)+(R5) \to R2$ {продвижение указателя текущего элемента массива результатов}.
9. $D+8$ $(R3)-(R5) \to R3$ {продвижение счетчика — выработка признака результата}.
10. $D+9$ $ZF=0$? Да — $\Rightarrow D+10$, нет — $\Rightarrow D+5$.

Использование косвенных адресов позволяет заменить команды переадресации на переадресацию неявную: вместо команд переработки команд используются команды продвижения указателей, что приводит, в частности, к уменьшению числа обращений к памяти — при переадресации переменная

команда извлекается из памяти и затем отправляется в память (три обращения к памяти на команду), а продвижение указателя не использует данных из памяти (одно обращение).

Автоинкремент/декремент

Цикл — один из главных технологических элементов программирования. Необходимость эффективной поддержки циклов — создание условий для удобного программирования, сокращения занимаемой памяти, повышения быстродействия — важная побудительная причина совершенствования системы команд и способов адресации.

Как показывает статистика, большинство циклов — короткие, тело цикла состоит из нескольких команд. В такой ситуации непроизводительные затраты места и времени — на выполнение команд переадресации и восстановления — оказываются значительными. Внесение в архитектуру некоторых изменений позволяет существенно сократить эти затраты.

Одно из таких изменений — введение в машины, использующие регистровую косвенную адресацию, новых ее разновидностей, которые называются "косвенной адресацией с последующим инкрементом" и "косвенной адресацией с предшествующим декрементом". Их сокращенные названия — "адресация с автоинкрементом" и "адресация с автодекрементом". Например, команда вида

$$((R1)+)\,p(-(R2)) \to R0$$

выполняется так:

1. $((\text{УК})) \to \text{РК}$.
2. $(\text{УК}) + 1 \to \text{УК}$.
3. Дешифрировать код операции.
4. $((R1)) \to r1$.
5. $(R1) + 1 \to R1$ {Инкремент}.
6. $(R2) - 1 \to R2$ {Декремент}.
7. $((R2)) \to r2$.
8. $(r1)\,p(r2) \to r3$.
9. $(r3) \to R0$.
10. Перейти к пункту 1.

Пусть до выполнения команды регистр $R1$ содержал адрес A, а регистр $R2$ — адрес B. При выполнении команды первый операнд будет извлечен из

ячейки A, а затем изменится содержимое регистра $R1$ на $A+1$ (знак "+" в команде указывает на инкремент после прочтения содержимого ячейки A). Что касается второго операнда, то сначала содержимое регистра $R2$ будет заменено на $B-1$ (знак "–" указывает на предварительный декремент содержимого ячейки B), а затем содержимое ячейки $B-1$ будет использовано в качестве операнда.

Использование команды с автоинкрементом позволяет сократить составленную нами программу.

1. D $(C_N) \to R3$ {начальное значение счетчика}.
2. $D+1$ $A+1 \to R1$ {установка указателя на первый элемент массива аргументов}.
3. $D+2$ $B+1 \to R2$ {установка указателя на первый элемент массива результатов}.
4. $D+3$ $(C_\gamma) \to R4$ $\{\gamma \to R4\}$.
5. $D+4$ $((R1)+) \times (R4) \to (R2)+$ {основные вычисления с одновременным продвижением указателей текущих элементов массивов}.
6. $D+5$ $(R3)-1 \to R3$ {продвижение счетчика — выработка признака результата}.
7. $D+6$ $ZF=0$? Да — $\Rightarrow D+10$, нет — $\Rightarrow D+5$.

Косвенная адресация с автоинкрементом/декрементом позволяет вообще не иметь специальных команд переадресации — ни явной, ни неявной. Здесь переадресация совмещается с самой переменной командой (ни одного обращения к памяти для переадресации).

Недостатком такого способа переадресации является то, что продвижение всегда происходит только на некоторые стандартные для данной машины единицы данных: иногда только на байты, иногда — на байты и слова. В коде операции каждой команды содержится признак, указывающий на формат операндов (байт, слово). Используя этот признак, команды с автоинкрементом/декрементом продвигают указатель на один байт или на одно слово (два или четыре байта, в зависимости от машины). Но байт и слово не исчерпывают всех единиц данных, да и выборка из массивов не обязательно ведется подряд. Например, может потребоваться обработка каждого десятого байта.

Стек

Стек является важной разновидностью косвенной адресации с автоинкрементом/декрементом. *Стек* — часть оперативной памяти, доступ к которой организован специальным образом. Принцип организации стековой памяти тот

же, что у рожка автомата или магазина пистолета (между прочим, стек называют также магазином, хотя сейчас этот термин почти вышел из употребления). Патроны в магазин заряжаются только сверху и извлекаются тоже только сверху. Невозможно извлечь или зарядить патрон в середину магазина. Для стека выделяется некоторый участок памяти, скажем, от ячейки A до ячейки B. Ячейка B, имеющая наибольший адрес, объявляется "дном" стека, ячейка A, с наименьшим адресом, объявляется вершиной стека. Стек может быть и "бездонным", как на рис. 4.25. Во всяком случае, за дном стека программа обычно не следит, перекладывая проблему исчерпания стека на операционную систему. Но это за рамками нашей темы и мы будем считать стек бездонным. А за вершиной стека следит специальный стековый регистр. Обозначим его R_{SP} (stack pointer — указатель стека). Если $(R_{SP}) = A$, то ячейка A — вершина стека.

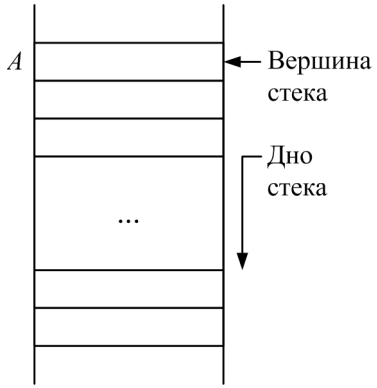


Рис. 4.25. Вершина и дно стека

Действует стек следующим образом. Считается, что все ячейки стека от вершины и ниже заполнены данными. Если нам надо извлечь данное из стека, мы можем взять то, на которое указывает стековый регистр. После извлечения данного из стека указатель вершины автоматически перемещается "вниз", т. е. к его содержимому прибавляется единица. Если мы решаем поместить в стек новое данное, указатель вершины перемещается вверх (из содержимого стекового регистра вычитается единица, теперь он указывает на первую свободную ячейку выше старой вершины), а затем данное засылается в ячейку, адрес которой хранит стековый регистр.

В системе команд машины имеется группа специальных стековых команд. Обычно это команды записи из памяти или из регистра в стек и чтения из стека в память или в регистр. Одним из операндов в них служит стековый

регистр, вторым — любой регистр или ячейка памяти. Стековые команды записи и чтения отличаются от других команд записи и чтения кодами операции, и поэтому стек в качестве одного из операндов в них подразумевается. Будем обозначать коды стековых операций через P_{SP}. Тогда стековые команды можно изобразить в виде $P_{SP}\ R$ или $P_{SP}\ C$.

Стековая команда "запись в стек из регистра" выполняется следующим образом:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $(R_{SP}) - 1 \rightarrow R_{SP}$.
5. $(R1) \rightarrow (R_{SP})$.
6. Перейти к пункту 1.

Стековая команда "запись в стек из памяти" (говорят "выталкивание из стека") отличается только пунктом 5: $(C) \rightarrow (R_{SP})$.

Стековая команда "выталкивание в регистр из стека":

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $((R_{SP})) \rightarrow R$.
5. $(R_{SP}) + 1 \rightarrow R_{SP}$.
6. Перейти к пункту 1.

В общем, это похоже на косвенную регистровую адресацию с автоинкрементом/декрементом. Только здесь не надо явно указывать регистр — он подразумевается. И еще: если мы извлекаем данное из стека, то его вершина продвигается только вниз, а если помещаем данное в стек, то его вершина продвигается только вверх. А в командах с автоинкрементом/декрементом в любом случае возможно продвижение в любую сторону.

Важно отметить, что пространство стека — это обычные ячейки памяти и их можно использовать в не стековых командах как обычные. То есть если мы занесем A в указатель стека, это никак не повлияет на исполнение не стековых команд. Можно, например, изменить содержимое ячейки A, для стека она все равно останется вершиной, пока мы не занесем в стековый регистр новое значение.

Более того, стековый регистр может участвовать в любых арифметико-логических операциях как обычный регистр общего назначения. Например, команда $P\ R_{st}\ R2$ выполняется по обычной схеме — $(R_{SP}) * (R2) \to R2$.

Для обычных, не стековых команд, стек как бы не существует. Для них все ячейки памяти одинаковые, в том числе и те, что используются стеком. Для стековых же команд существует только одна ячейка — вершина стека, которая передвигается по памяти.

Использование стековых команд предоставляет меньшую гибкость, чем использование команд с автоинкрементом/декрементом, но стековые команды короче, т. к. стековый регистр в команде не указывается.

Индексный регистр

Более гибким способом неявной переадресации является использование индексных регистров. В ранних машинах в процессор вводился специальный регистр, предназначенный для переадресации. Он назывался индексным — ИР. В команду были введены специальные признаки, указывающие, какие адреса используют индексный регистр — $\sigma 1_{и}$, $\sigma 2_{и}$, $\sigma 3_{и}$. Команда в такой машине исполняется по схеме:

$$(E1)\,p\,(E2) \to E3, \text{ где } Ei = Ai + \sigma i_{и} \times (\text{ИР}).$$

Ei называют исполнительным или эффективным адресом. Предыдущий пример (напомним его: требуется помножить N элементов массива, начинающегося в ячейке $A+1$, на константу $\gamma = (C\gamma)$ и результаты поместить в массив, начинающийся в ячейке $B+1$; $(C_N) = N$, $(C_1) = 1$) в машине с индексным регистром можно запрограммировать так:

1. D $(C_N) \to H$ {начальное значение счетчика}.
2. $D+1$ $(C_1) \to$ ИР {начальное значение индекса = 1}.
3. $D+2$ $(A + (\text{ИР}) \times \sigma 1_{и}) \times (C\gamma) \to B + (\text{ИР}) \times \sigma 3_{и}$ {основные вычисления}.
4. $D+3$ $(\text{ИР}) + (C_1) \to$ ИР {продвижение индекса}.
5. $D+4$ $(H) - (C_1) \to H$ {продвижение счетчика — выработка признака результата}.
6. $D+5$ $ZF = 0$? Да — $\Rightarrow D+6$, нет — $\Rightarrow D+2$.

Такой способ неявной переадресации позволяет выполнять переадресацию на любое число байтов.

Следует, однако, иметь в виду, что существует потребность в более сложной переадресации. Рассмотрим пример.

Пусть $X_1 = (A+1)$, $X_2 = (A+2)$, ..., $X_{30} = (A+30)$ и $Y_1 = (B+1)$, $T_2 = (B+2)$, ..., $Y_{30} = (B+30)$.

Пусть требуется вычислить $Z_1 = X_1 \times Y_3$, $Z_2 = X_2 \times Y_6$, ..., $Z_{10} = X_{10} \times Y_{30}$ с размещением Z_i в ячейках $C+1$, $C+2$, ..., $C+10$.

Очевидно, переменная команда в цикле должна первоначально иметь вид $(A+1)\times(B+3)\to C$, затем $(A+2)\times(B+6)\to C+2$ и т. д. с увеличением на каждом шаге первого и третьего адресов на 1, а второго — на 3. Один индексный регистр не может обеспечить одновременную переадресацию с разным шагом, а автоинкремент вообще не может переадресовывать на 3.

Еще сложнее обстоит дело в случае двойного цикла. Пусть в памяти хранятся массивы $X_1 = (A+1)$, $X_2 = (A+2)$, ..., $X_N = (A+N)$ и $\gamma_1 = (C_\gamma + 1)$, $\gamma_2 = (C_\gamma + 2)$, ..., $\gamma_M = (C_\gamma + M)$.

Требуется вычислить все $Y_{ij} = \gamma_j X_i$.

Очевидно, надо организовать двойной цикл. Основные вычисления будут выполняться одной командой с переменными адресами. Первые N раз цикл надо выполнять с командами вида $(A+i)\times(C_\gamma+1)\to B+i$ ($i = 1, 2, ..., N$), затем — N раз с командами вида $(A+i)\times(C_\gamma+2)\to B+N+i$ и т. д.

Получается, что в команде должны одновременно увеличиваться на 1 первый и третий адреса (параметр i) и, изредка, каждый раз после N исполнений, второй увеличиваться на 1, а первый — уменьшаться на N.

Для облегчения обслуживания сложных циклов были сконструированы машины с несколькими индексными регистрами.

Команды в таких машинах имеют следующий вид (рис. 4.26).

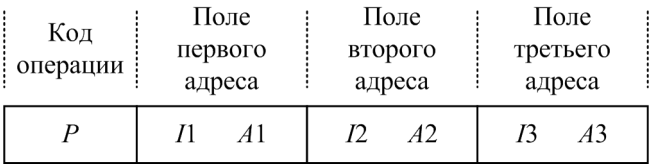


Рис. 4.26

В поле каждого из адресов команды располагаются два числа. В случае работы с блоками данных или команд второе число — адрес начала блока, пер-

вое — номер индексного регистра, содержащего смещение нужного элемента блока относительно начала (рис. 4.27).

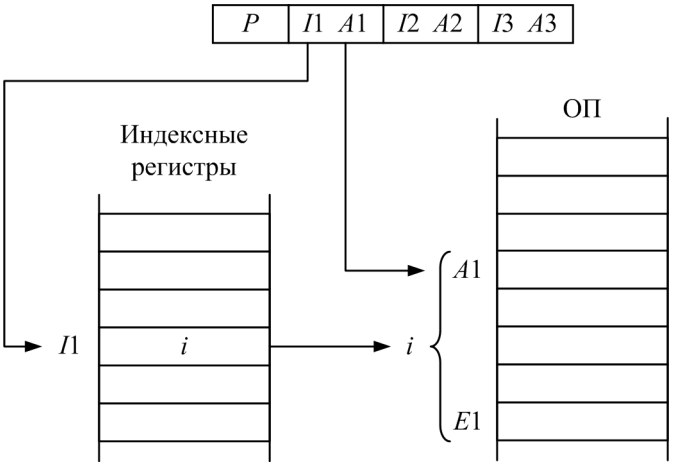


Рис. 4.27. Схема образования адреса с использованием индексного регистра

Если каждый элемент располагается в одной ячейке, то содержимое индексного регистра — порядковый номер (индекс) элемента, при условии, что в ячейке $A1$ находится элемент номер ноль. Исполнительные или эффективные адреса в такой команде вычисляются по формуле:

$$Ei = Ai + (Ii).$$

Составные адреса являются обобщением и развитием предыдущих способов адресации. Если $(Ii) = 0$, то фактически адрес прямой. Если $Ai = 0$, то адрес — регистровый косвенный.

В приведенном выше примере с двойным циклом основные вычисления требуют трехадресной команды, в которой используются три различных индексных регистра.

Конечно, не редкость и еще более сложные циклы: тройные, четверные и т. д. Для них требуется еще большее количество индексных регистров. В общем, можно сказать так. Чем больше индексных регистров, тем проще организовывать сложные циклы. Но, с другой стороны, чем больше индексных регистров, тем сложнее, а следовательно, дороже машина. Значит, надо искать компромисс. Мы видели, что можно обходиться вообще без индексных регистров, ясно, что можно обойтись и малым их числом. Только программы будут получаться посложнее. Однозначно определить оптимальное число индексных регистров вряд ли возможно. Поэтому существуют машины с разным числом ИР: например, два, восемь, шестнадцать.

Обычно в машине наряду с индексными имеются и числовые регистры, предназначенные для хранения данных или адресов данных. Тогда в машине имеются также команды вида, представленного на рис. 4.28.

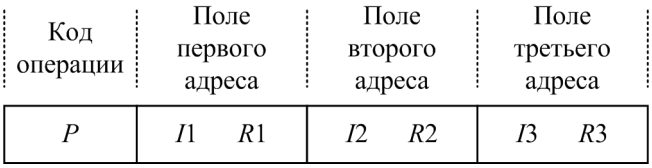


Рис. 4.28

При их исполнении формируются исполнительные адреса

$$Ei = (Ri) + (Ii).$$

Схема их формирования — на рис. 4.29.

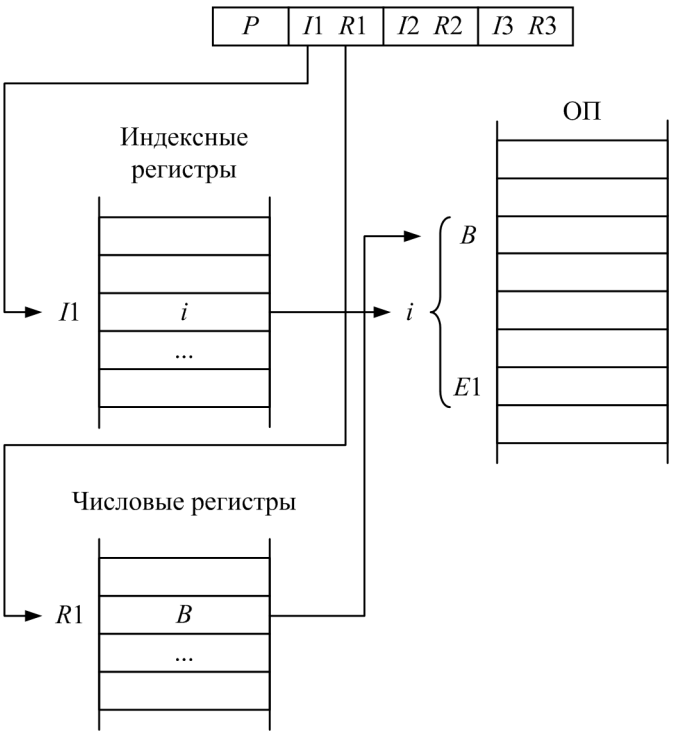


Рис. 4.29. Формирование адреса командами, изображенными на рис. 4.28

Часто в машине одни и те же регистры могут использоваться и как индексные, и как числовые. Их тогда называют регистрами общего назначения.

4.2.3. Сложные команды управления циклом

Команда управления + продвижение индекса

Повторимся: обычно тело цикла состоит всего из нескольких команд и удельный вес каждой команды значителен. Поэтому стоит подумать об экономии. Возможность для этого заложена в том факте, что двух- или трехадресные команды передачи управления имеют лишь один существенный адрес. Можно оставшиеся адреса использовать для встраивания в команду дополнительных функций. Команду передачи управления можно совместить с увеличением индекса, если включить в систему команд команду вида $P\ R1\ A2$, выполняющуюся по схеме:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать код операции.
4. $(R1)+1 \rightarrow R1$.
5. Если ((РПр) AND $(N_p \neq 0)$, то $A2 \rightarrow$ УК.
6. Перейти к пункту 1.

Это дает экономию: длина тела цикла на команду меньше, обращений к памяти на одно меньше.

Но тогда счетчик придется наращивать отдельной командой.

Команда управления + счетчик

Счетчик — элемент многих циклов. Имеет смысл добавить в систему команд специальные команды управления циклом со счетчиком. То есть наряду с универсальными командами условной передачи управления, пригодными как для простого разветвления, так и для управления циклом, включить в систему команд специализированные команды управления циклом, одновременно управляющие счетчиком.

Например, в двухадресной машине может иметься команда вида $P\ R1\ A2$, которая выполняется следующим образом:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать код операции.

4. $(R1)-1 \to R1$.

5. Если $(R1) \neq 0$, то $A2 \to$ УК.

6. Перейти к пункту 1.

Этой командой иногда можно пользоваться и для продвижения индекса. Если вести обработку массива в обратном порядке — от больших индексов к меньшим, то тот же регистр можно использовать и в качестве индексного.

Команда управления + индексирование + счетчик

Часто встречаются задачи, в которых необходимо выбирать данные из массива, например, только по четным адресам или вообще с некоторым шагом h. Для этой цели в некоторых машинах существуют даже еще более сложные команды. Там команда вида $P\ R1\ R2\ A3$ выполняется по такой схеме:

1. ((УК)) $\to$ РК.

2. (УК) + 1 $\to$ УК.

3. Дешифрировать код операции.

4. $(R1)+(R2) \to R1$.

5. Если $(R1)=(R2+1)$, то $A3 \to$ УК.

6. Перейти к пункту 1.

Здесь, кроме регистра $R1$, используется пара подряд расположенных регистров — $R2$ и $R2+1$: $R2$ — для шага переадресации h, $R2+1$ — для конечного значения индекса.

Это позволяет сократить тело цикла до минимума — исчезают отдельные команды, связанные с переадресацией и со счетчиком. Правда, нисколько не сокращается число команд подготовки цикла, но это не так важно — ведь они выполняются всего один раз.

Посмотрим, как выглядит программа для самого первого примера. Напомним его.

Пусть переменные X_1, X_2, ..., X_n хранятся в последовательных ячейках памяти: $X_1=(A+1)$, $X_2=(A+2)$, ..., $X_n=(A+N)$. Пусть $\gamma=(C_\gamma)$.

Пусть требуется вычислить $Y_1=\gamma X_1$, $Y_2=\gamma X_2$, ..., $Y_n=\gamma X_n$ и разместить результаты в последовательности ячеек — $B+1$, $B+2$, ..., $B+N$.

1. $D\ \ (C_N) \to R2+1$ {конечное значение индекса i}.

2. $D+1\ \ 1 \to R1$ {начальное значение индекса i}.

3. $D+2\ \ 1 \to R2$ {шаг индекса}.
4. $D+3\ \ (A+1+(R1)) \times C_\gamma \to B+1+(R1)$ {основные вычисления; используется команда D вида: $P\ R1, A+1\, C_\gamma\ R1, B+1$}.
5. $D+4\ \ (R1)+(R2) \to R1$, $(R2+1)=(R1)$? Да — $\Rightarrow D+5$, нет — $\Rightarrow D+3$ {продвижение индекса, управление циклом; используется команда вида: $P\ R1\ R2\ D+3$}.

Тело цикла состоит всего из двух команд: основные вычисления и управление циклом.

Казалось бы, это предел. Но нет! Существуют машины, для которых возможен цикл из одной команды.

В некоторых машинах регистры специализированные. Часть — индексные, часть — арифметические (числовые). Отдельные регистры — счетчики, отдельные — аккумуляторы и т. д. Правда, обычно они могут использоваться и как универсальные. То есть, например, любой регистр можно использовать как счетчик в цикле, но по умолчанию используется конкретный, всегда один и тот же регистр. В этих машинах имеется небольшой набор команд (в пределах десятка), которые могут составлять в одиночку тело цикла. Обычно это команды ввода и вывода, сравнения пересылки и несколько подобных. Их называют цепочечными командами.

Такая команда вообще не имеет адресов и состоит только из кода операции. Например, команда сравнения цепочек выполняется примерно так:

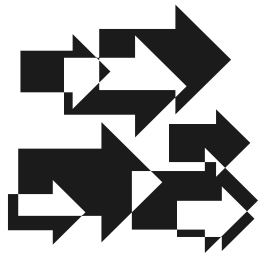
1. ((УК)) $\to$ РК.
2. (УК) + 1 $\to$ УК.
3. Дешифрировать код операции.
4. Если $(R_C)=0$, то перейти к пункту 1.
5. $(R_C)-1 \to R_C$.
6. $((R_{SI}))-((R_{DI})) \to ?$.
7. Если $(\text{РПр})=0$, то перейти к пункту 1.
8. $(R_{SI})+1 \to R_{SI}$, $(R_{DI})+1 \to R_{DI}$.
9. Перейти к пункту 4.

Эта команда сравнивает элементы двух массивов. Подразумевается, что адрес начала первого предварительно помещен в регистр SI, адрес второго — в регистр DI, а в регистр C предварительно помещена длина этих массивов.

Команда последовательно сравнивает первые, затем вторые и т. д. элементы массивов до тех пор, пока не найдутся одинаковые или не кончатся массивы.

Заметьте, здесь в одной команде выполняются и продвижение счетчика, и переадресация, и два сравнения на конец цикла. Но ценность такой команды не только в этом. В предыдущем примере тело цикла состояло из двух команд и при каждом из N повторений команды снова и снова вызывались из памяти. Всего $2N$ обращений только за командами. Цепочечная же команда вызывается только один раз.

ГЛАВА 5



Подпрограммы и ввод/вывод

5.1. Подпрограммы

Обычно *подпрограммы* (ПП) составляются как универсальные программы, способные взаимодействовать с любыми главными программами. Составление ПП обычно предшествует составлению главной программы. Более того, имеются библиотеки ПП, и составитель главной программы просто пользуется готовыми ПП. Иногда библиотечные ПП называют стандартными ПП. Естественно, есть правила работы с ПП библиотеки, и, конечно, составители библиотек стремятся сделать эти правила как можно проще. Эти правила называют соглашениями о ПП. Разумеется, каждый программист может, наряду с библиотечными, пользоваться ПП собственного изготовления, которые не обязаны соответствовать соглашениям. Обычно ПП хранятся во внешней, медленной памяти и извлекаются оттуда по мере надобности. Они организованы так же, как любая библиотека, с помощью каталога. В каталоге каждой ПП поставлена в соответствие некоторая запись. В записи содержится имя ПП, ее адрес хранения, длина и некоторая другая информация.

5.1.1. Схема взаимодействия ПП с главной программой

Пусть, например, требуется составить программу для вычисления значения z по формуле

$$z = f(x)\sin x + g(x).$$

В некоторых компьютерах операция $\sin x$ реализована как элементарная, выполняемая одной командой. Во многих — такой операции нет и необходимо вычислять эту функцию программно.

При малых x можно воспользоваться отрезком ряда

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \ldots,$$

но прежде надо привести аргумент к интервалу $(0, 2\pi)$, затем — $(0, \pi/2)$, далее, используя формулы для половинных углов, может быть, неоднократно, свести задачу к нахождению синуса от малого аргумента. В общем, так или иначе, для нахождения синуса надо написать программу примерно из 30 команд.

Целесообразно составить две программы: подпрограмму, вычисляющую значение функции $\sin x$, и главную программу, вычисляющую значение z и использующую подпрограмму. В нужном месте главной программы должна стоять команда безусловного перехода к подпрограмме, а в конце подпрограммы — команда безусловного перехода к главной (рис. 5.1).

Команду в ячейке A называют командой перехода к подпрограмме, вызовом подпрограммы. Это разновидность команд безусловного перехода. D — точка входа в подпрограмму, E — точка выхода. Чаще всего точка входа — первая команда ПП, но это не обязательно.

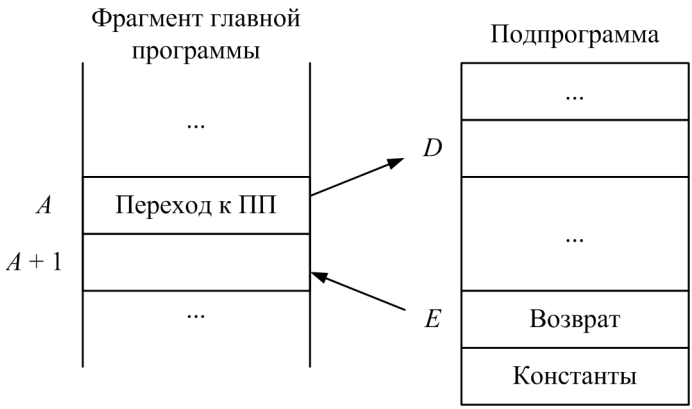


Рис. 5.1. Схема взаимодействия главной программы и подпрограммы

Точка выхода чаще всего не последняя ячейка подпрограммы — как и у всякой программы в ее конце обычно находятся нужные ей константы из тех, что нецелесообразно записывать в качестве непосредственных адресов. Более того, точка выхода может находиться даже выше точки входа.

После выполнения подпрограммы нужно продолжить работу главной программы, так что в ячейке E должна находиться команда безусловного перехода к основной программе в ячейку $A+1$. Ее называют командой возврата.

Рассмотрим более сложный пример. Пусть z имеет вид

$$z = f(x)\sin\alpha x + g(x)\sin\beta x .$$

Здесь целесообразность применения ПП очевидна. Конечно, надо использовать одну подпрограмму для вычисления как $\sin\alpha x$, так и $\sin\beta x$ и для этого обратиться к подпрограмме два раза, из двух мест, как на рис. 5.2.

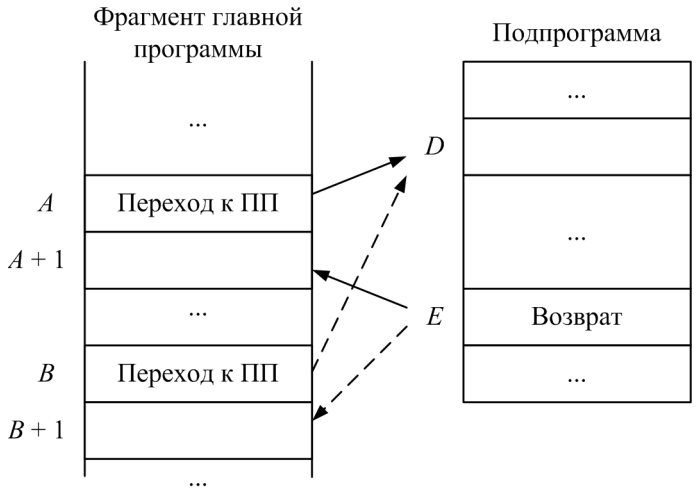


Рис. 5.2. Обращение к ПП из двух мест

Разумеется, перед каждым обращением нужно правильно задать аргумент (принято выражение "передать аргумент"): в первый раз — αx, во второй — βx.

Еще более сложный пример:

$$z = \left(a_1x^2 + a_2x + a_3\right)\mathrm{Si}(\alpha x) + \left(b_1x + b_2\right)\mathrm{Si}(\beta x), \text{ где } \mathrm{Si}(y) = \int_0^y \frac{\sin t}{t}.$$

$\mathrm{Si}(y)$ — интегральный синус. Для приближенного вычисления определенного интеграла можно воспользоваться, например, формулой Симпсона:

$$\int_a^b F(t)dt \approx \frac{b-a}{6}\left[F(a) + 4F\left(\frac{a+b}{2}\right) + F(b)\right].$$

Получается, что главной программе нужна ПП, вычисляющая $\mathrm{Si}(y)$, которой нужна ПП, вычисляющая $\sin(t)$. ПП $\mathrm{Si}(y)$ каждый раз, когда к ней обращается главная программа, трижды вычисляет значение аргумента t и обраща-

ется с ним к ПП $\sin(t)$. Схема взаимодействия главной программы и подпрограмм представлена на рис. 5.3.

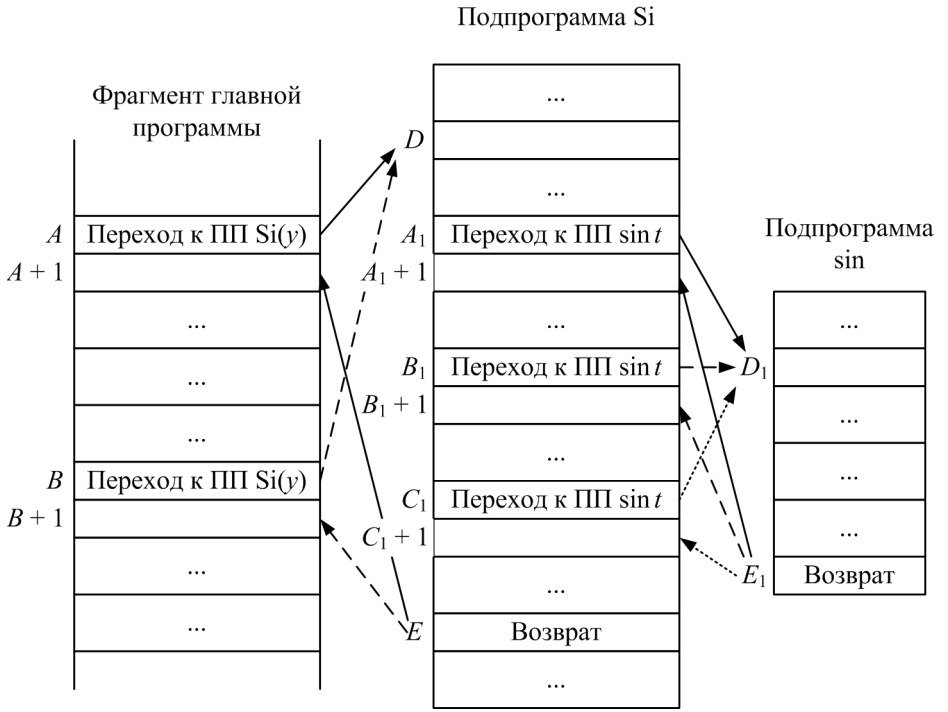


Рис. 5.3. Схема взаимодействия вложенных подпрограмм

В связи с использованием ПП возникает ряд проблем, которые носят названия:

- ❒ вызов ПП и возврат;
- ❒ передача параметров;
- ❒ сохранение регистров;
- ❒ настройка по параметрам;
- ❒ настройка по месту.

Обсудим их последовательно.

5.1.2. Вызов ПП и возврат

В ячейках A и B должны быть команды перехода по адресу D, с этим все ясно. А вот команда возврата в ячейке E — переменная. Когда главная ПП

будет выполняться после вызова из ячейки A, в ячейке E должна стоять команда перехода по адресу $A+1$, а после вызова из ячейки B — по адресу $B+1$. А если представить, что этот участок главной программы должен исполняться в цикле, то окажется, что в ячейке E постоянно меняется адрес возврата.

Как подпрограмма узнает, куда ей надо передавать управление? Очевидно, информация об адресе возврата должна быть передана из главной программы перед вызовом подпрограммы. Разработано несколько приемов организации возврата.

Засылка в ПП команды возврата

Пусть двухадресная команда безусловного перехода по адресу D имеет вид БП ~ D (первый адрес на выполнение команды не влияет). Пусть команды $(\gamma) \to E$ и $(\gamma+1) \to E$ — пересылки содержимого ячеек γ и $\gamma+1$ в ячейку E, а в ячейках γ и $\gamma+1$ заранее заготовленные константы в виде команд безусловного перехода: $(\gamma) = \text{БП} \sim A+1$, $(\gamma+1) = \text{БП} \sim B+1$.

Тогда тот же фрагмент программы с подпрограммой может выглядеть так, как показано на рис. 5.4.

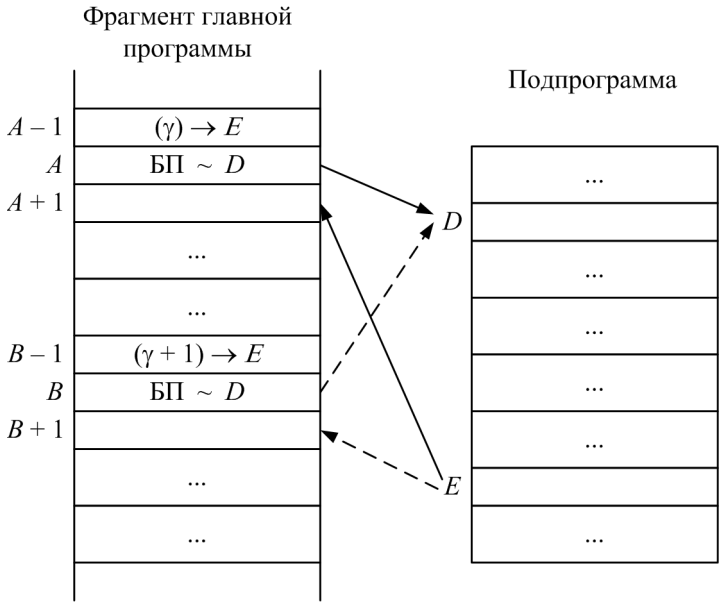


Рис. 5.4. Схема с засылкой команды возврата

Такой способ организации связи главной программы с ПП требует на каждое обращение по три ячейки ($A-1$ и γ — для первого, $B-1$, B и $\gamma+1$ — для второго), исполнения двух команд и четырех обращений к памяти. Естественно, возникает идея совместить в одной команде переход и передачу информации о точке возврата.

Пожалуй, самое простое решение проблемы организации возврата — объединить команды из ячеек $A-1$ и A в одну команду. В систему команд машины вводится специальная двухадресная команда перехода к подпрограмме (обозначение кода операции — ПП) вида ПП E D.

Она выполняется следующим образом.

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.
4. "БП 0 (УК)" $\rightarrow$ E.
5. D $\rightarrow$ УК.
6. Перейти к пункту 1.

Таким образом, при обращении к ПП из ячейки A пункт 4 сформирует и зашлет в ячейку E команду вида БП 0 $A+1$, что обеспечит возврат из подпрограммы в ячейку $A+1$, а при обращении из ячейки B в ту же ячейку будет записана команда БП 0 $B+1$, так что и в этом случае взаимодействие ПП с основной программой будет правильным.

Особенность команды ПП E D: если $E=0$, то пункт 4 из приведенного описания игнорируется.

Такая организация взаимодействия требует одной ячейки, одной команды, двух обращений к памяти.

При использовании этой команды перехода к ПП составитель главной программы должен знать адрес точки входа в ПП и адрес точки выхода. Впрочем, знание точки выхода можно исключить, если использовать такой прием (рис. 5.5).

Последняя команда ПП (в ячейке E) — безусловная передача управления в ячейку, непосредственно предшествующую точке входа — $D-1$. Теперь программисту не надо знать, где расположена точка выхода. Достаточно знать адрес точки входа — D. Точка выхода — ячейка $D-1$. И команда обращения к ПП выглядит так: ПП $D-1$ D.

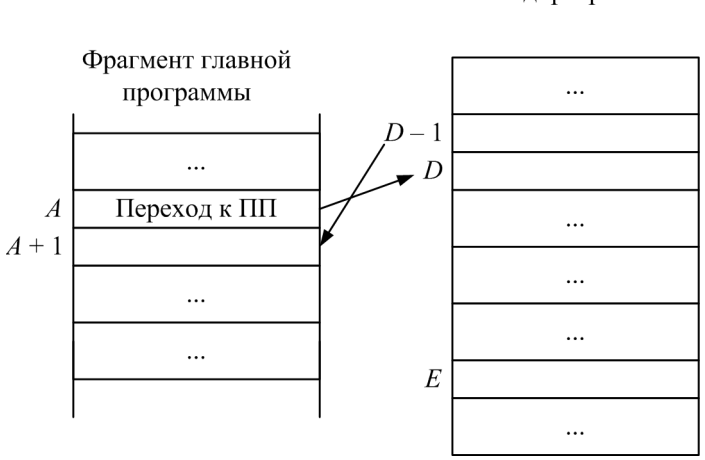


Рис. 5.5. Схема с командой возврата в начале подпрограммы

Сохранение адреса возврата в регистре

В систему команд вводятся две команды — команда вызова и команда возврата. Команда вызова ПП имеет вид ПП $R1$ D, а команда возврата — БП 0 $R1$.

Первая из них исполняется по схеме

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.
4. (УК) $\rightarrow$ $R1$.
5. D $\rightarrow$ УК.
6. Перейти к пункту 1.

Вторая:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.
4. ($R1$) $\rightarrow$ УК.
5. Перейти к пункту 1.

Схема взаимодействия программы и ПП в этом случае такая, как на рис. 5.6.

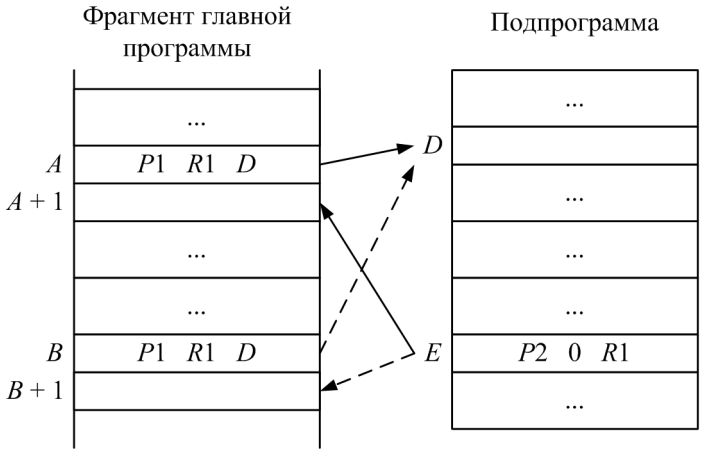


Рис. 5.6. Схема с командой возврата в регистре

Часто используется разновидность команды перехода к подпрограмме — переход по относительному адресу. Схема ее исполнения отличается от приведенной выше лишь пунктом 5: $(\text{УК}) + D \to \text{УК}$.

Конечно, команда перехода к ПП может быть и в формате "регистр — регистр", иметь вид ПП $R1$ $R2$ и передавать управление по регистровому адресу. Для перехода к ПП надо предварительно занести в $R2$ число D.

В машинах с такой командой вызова ПП нет специальной команды возврата. Команде вызова придается особенность: если в ее первом адресе стоит ноль, то адрес возврата не пишется никуда, а если во втором адресе стоит ноль, то адрес возврата запоминается в регистре $R1$, но управление не передается никуда, точнее, следующей команде. Словом, команда действует по схеме, изображенной на рис. 5.7.

Поэтому в качестве команды возврата можно использовать команду с тем же кодом операции, что и у команды вызова: ПП 0 $R1$.

Нужно только, чтобы совпадали первый адрес команды вызова и второй — команды возврата.

Обратите внимание: в качестве $R1$ можно выбрать любой из регистров общего назначения, но в командах обращения и в команде возврата должен использоваться один и то же регистр. То есть составитель ПП должен сообщить составителю главной программы, какой регистр он отвел для адреса возврата. Обычно это один и тот же регистр для всех ПП одной библиотеки (номер этого регистра — часть соглашений о ПП).

Использование этого приема требует одного обращения к памяти, и программисту не надо знать точку выхода.

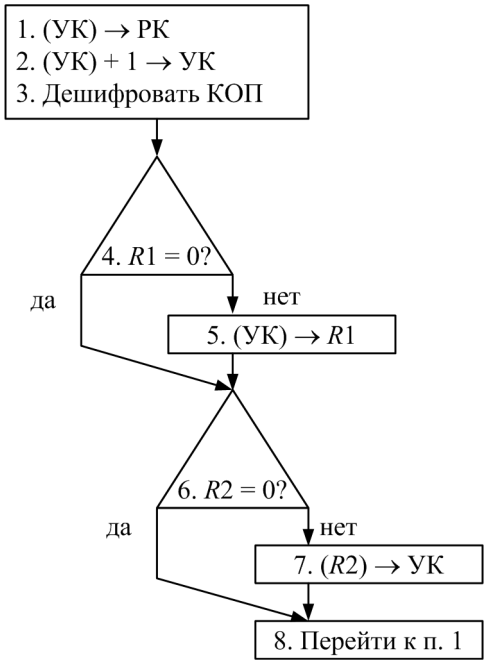


Рис. 5.7. Команда возврата, как частный случай команды перехода к ПП

Однако есть и недостаток — возникает проблема вложенных ПП. Она заключается в следующем. Пусть в соглашении о ПП для сохранения адреса возврата выделен регистр $R(1)$. (Здесь и далее $R(1)$ — это регистр № 1, в то время как $R1$ — любой регистр.) Главная программа, обратившись к ПП первого уровня, засылает адрес возврата — $A+1$ в $R(1)$ (рис. 5.8).

Затем ПП первого уровня, в свою очередь, обращается к ПП второго уровня, передает ей управление и засылает свой адрес возврата — A_1+1 в $R(1)$. ПП второго уровня, выполнив свою работу, передаст управление по адресу, содержащемуся в регистре $R(1)$, т. е. вернет управление ПП первого уровня в ячейку A_1+1. Когда ПП первого уровня закончит работу, последней будет выполняться команда возврата, находящаяся в ячейке E. Она передаст управление по адресу, хранящемуся в $R(1)$. Но что в этом регистре? То, что заслали в последний раз — A_1+1. А надо передать управление в ячейку $A+1$. Но этот адрес утерян. И мы получим зацикливание на участке от A_1+1 до E.

Эта проблема решается так: перед обращением к подпрограмме второго уровня подпрограмма первого уровня копирует содержимое $R(1)$ в специаль-

но выделенную в ее теле ячейку. А после возврата из подпрограммы второго уровня копия снова засылается в $R(1)$. Если подпрограмма второго уровня сама обращается к подпрограмме третьего уровня, то и она сохраняет копию $R(1)$ в своем теле. Таким образом, при вложенности любой кратности мы обеспечиваем правильное взаимодействие.

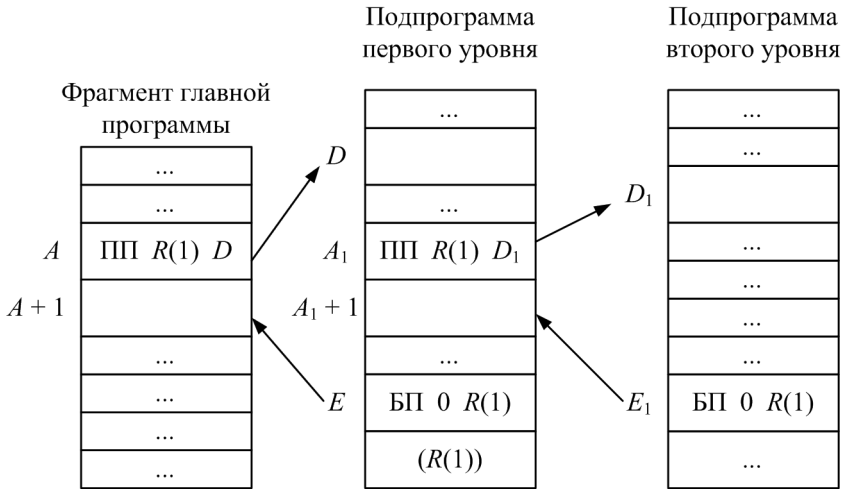


Рис. 5.8. Ошибка при использовании регистра для адреса возврата

Использование стека

Использование стека снимает проблему вложенных подпрограмм. В систему команд компьютера вводится пара команд:

1. Переход к подпрограмме — ПП D.
2. Возврат — $БП_{SP}$.

Первая команда одноадресная. Подразумевается, что адрес возврата сохраняется в стеке. Вторая — не имеет адресов. Подразумевается, что адрес безусловной передачи управления хранится в стеке.

Алгоритмы их исполнения приведены далее.

ПП D:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.

4. (УК) $\rightarrow -(SP)$.
5. $D \rightarrow$ УК.
6. Перейти к пункту 1.

$БП_{SP}$:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.
4. $(SP)+ \rightarrow$ УК.
5. Перейти к пункту 1.

Содержимое стека и положение его указателя при взаимодействии главной программы с подпрограммами изображены на рис. 5.9.

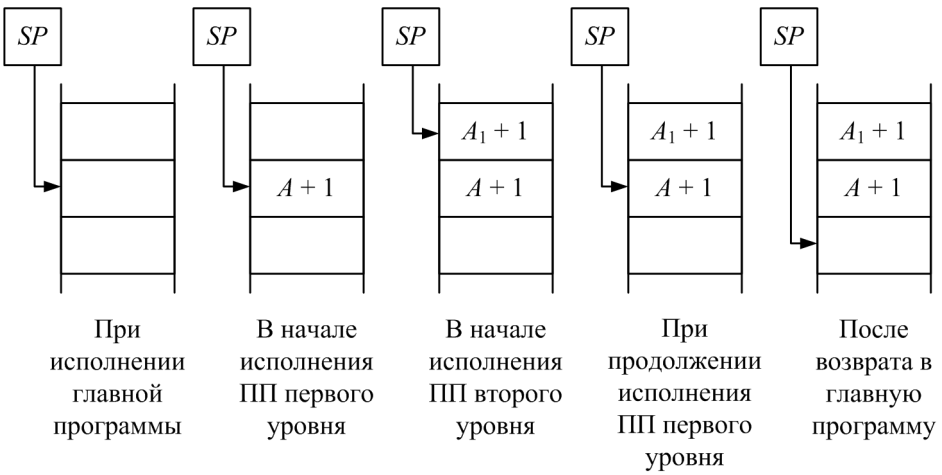


Рис. 5.9. Стек и его указатель
при выполнении главной программы и подпрограмм

При использовании стека для связи с подпрограммой происходят два обращения к памяти. Команда возврата также обращается к памяти два раза (два описанных прежде приема требовали лишь одного обращения к памяти). В общем, этот прием — не самый экономный. Но самый простой и сейчас является самым распространенным.

5.1.3. Передача параметров

Стандартные ячейки или регистры

Самый простой способ передачи параметров главной программой подпрограмме — использование стандартных входных и выходных ячеек или регистров. ПП вычисляет функцию (в нашем случае $\sin t$) от аргумента, находящегося в некотором регистре (или в ячейке), оговоренном в соглашении о ПП, например, в регистре 1 и помещает результат тоже в стандартный регистр, например, 2. Главная программа должна перед каждым переходом к ПП засылать нужный аргумент в стандартный регистр (стандартную ячейку), а после возврата из ПП использовать результат ее работы, находящийся в другом стандартном регистре.

На рис. 5.10 демонстрируется эта идея. В главной программе перед переходом к ПП выполняется некоторая операция (символ "*" означает произвольную операцию), результат которой засылается в регистр 1. В частности это может быть просто команда пересылки. Первая команда ПП использует содержимое этого регистра. Перед возвратом в главную программу ПП выполняет операцию, результат которой засылается в регистр 2. Главная программа использует этот результат.

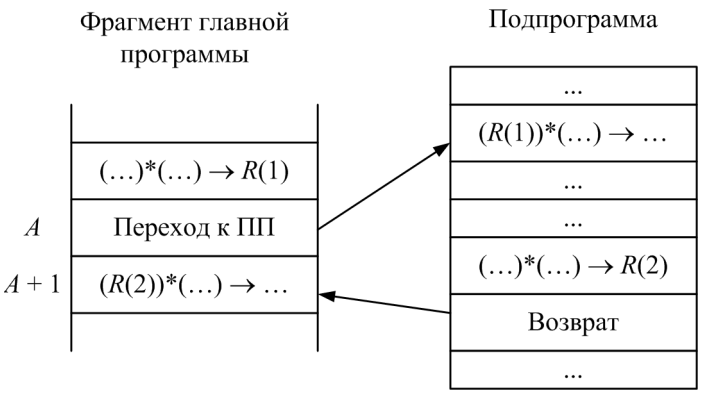


Рис. 5.10. Передача параметров через стандартные регистры

Замечания

- На рисунке команды главной программы (засылающая аргумент и использующая результат) помещены вплотную к команде перехода к ПП условно. Они могут так располагаться, и чаще всего подобным образом и располагаются, но это не обязательно. Команда, использующая результат, вообще может быть не одна.

- Команда ПП, использующая аргумент, также может быть не одна. Команда, засылающая результат, тоже не обязана располагаться непосредственно перед командой возврата.

Передача параметров через косвенный адрес

ПП может вычислять значение функции нескольких аргументов. Результатов тоже может быть несколько. Если, например, ПП вычисляет функцию двух аргументов, то можно использовать для передачи аргументов подпрограмме регистры 1 и 2, а результат помещать в регистр 3. Однако аргументов может быть вообще больше, чем регистров. Тогда аргументы передают ПП непосредственно из памяти. И тут действует другое соглашение. Оно требует, чтобы аргументы располагались в памяти в последовательных ячейках в определенном порядке (порядок определяет автор ПП и сообщает о нем тем, кто будет пользоваться его ПП в специальной инструкции). Результаты ПП разместит, если их много, также в памяти в последовательных ячейках. В этом случае главная программа должна сообщить подпрограмме адреса входной и выходной последовательностей. Это можно сделать через регистры, которые укажет в инструкции составитель ПП.

Пусть в инструкции указаны регистры 1 для последовательности аргументов и 2 для последовательности результатов. И пусть α и β — адреса последовательности аргументов и результатов соответственно. Тогда взаимодействие главной программы с ПП происходит так, как показано на рис. 5.11.

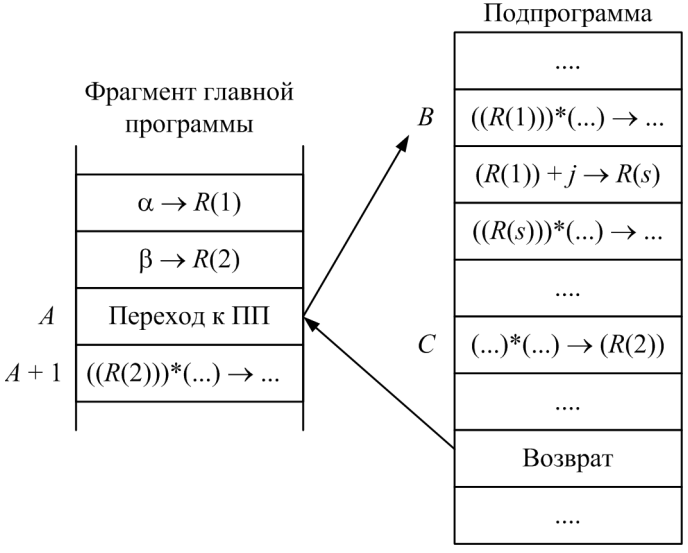


Рис. 5.11. Использование косвенной адресации для передачи параметров

В этом примере главная программа и ПП используют косвенную регистровую адресацию. Главная программа помещает в регистр 1 первый адрес последовательности или, можно сказать, ссылку на последовательность аргументов. Этот способ передачи параметров называют *передачей по ссылке*. В предыдущих примерах в регистр или в ячейку помещалось само значение параметра. Такой способ называется *передачей по значению*.

Возвращаясь к приведенному примеру, заметим: команда в ячейке *B* подпрограммы использует первый параметр из набора, хранящийся по адресу, который хранится в регистре 1. Предположим, по логике подпрограммы далее нужен параметр входного набора из ячейки $\alpha + j$. Следующие две команды помещают в регистр s адрес параметра, хранящегося в памяти на расстоянии j ячеек от первого, и используют его. Аналогично используются остальные параметры.

Команда, расположенная в ячейке C, засылает первый из результатов в первую ячейку последовательности. Для засылки второго и последующих результатов можно использовать прием, который применялся для извлечения входных параметров, а можно, как на рис. 5.11, воспользоваться автодекрементом.

Применяется также и двойная косвенная адресация. Предположим, что обрабатывается список объектов, каждый из которых имеет несколько атрибутов, значения которых записаны в списки и требуется последовательно извлекать и обрабатывать первый, затем второй и т. д. атрибуты сначала первого, затем второго объекта и т. д.

На рис. 5.12 A — адрес списка списков или адрес списка адресов списков.

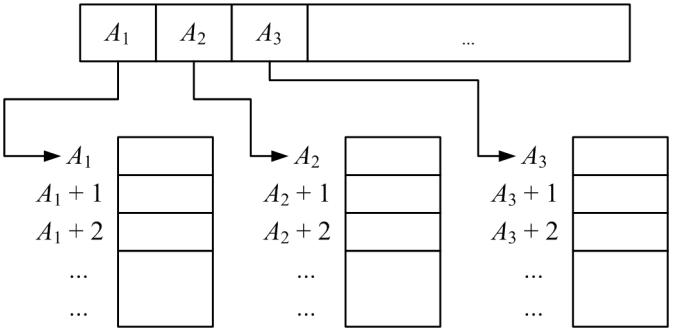


Рис. 5.12

Передача параметров через стек

Еще один способ передачи параметров, на сегодня самый распространенный, — через стек. Все параметры, которые нужно передать подпрограмме, или ссылки на них помещаются в стек в последовательности, которая задана составителем подпрограммы. Это делается перед обращением к подпрограмме. Ну а подпрограмма берет данные из стека или адреса данных из стека. Обычно этот метод используется вместе с методом сохранения адреса возврата в стеке. Тогда содержимое стека после перехода к подпрограмме выглядит так, как показано на рис. 5.13.

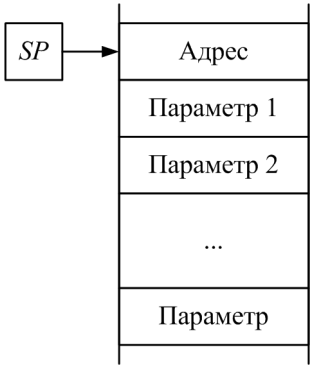


Рис. 5.13. Передача параметров через стек

Подпрограмма выбирает i-ый параметр из памяти по адресу $(SP)+i$, если он передается по значению, и по адресу $((SP)+i)$, если по ссылке.

Этот способ хорош тем, что он универсален, но проигрывает по быстродействию методу передачи через регистры в том случае, если параметров или ссылок не много. Главная программа должна данные или ссылки на них помещать в стек, т. е. пересылать в оперативную память. А подпрограмма должна извлекать их из памяти. Таким образом, при передаче параметров или ссылок через регистры экономится по два обращения к памяти на каждый параметр.

5.1.4. Сохранение регистров

Эта проблема заключается в следующем. ПП для размещения промежуточных результатов нужны регистры. Если главная программа перед обращением к подпрограмме держит в этих самых ячейках какие-то данные, то они будут подпрограммой уничтожены. Их надо как-то сохранять. Используются два приема. Первый — сохранение регистров в теле подпрограммы. Второй — сохранение в стеке. В этом случае к телу подпрограммы добавляется

несколько пустых ячеек — столько, сколько регистров использует подпрограмма. Первые же команды подпрограммы копируют содержимое используемых подпрограммой регистров в эти зарезервированные ячейки. В самом конце работы подпрограммы, перед возвратом в главную программу, выполняются команды загрузки в регистры старых значений (рис. 5.14).

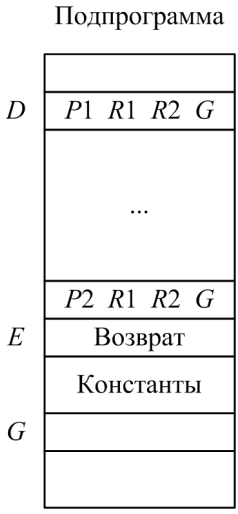


Рис. 5.14. Сохранение регистров в теле подпрограммы

Обычно в машинах имеются групповые команды копирования и загрузки регистров.

Команда копирования — $P1\ R1\ R2\ G$, команда загрузки — $P2\ R1\ R2\ G$. Первая копирует в ячейку G содержимое регистра $R1$, затем в ячейку $G+1$ — содержимое следующего регистра — $R1+1$ и т. д., кончая регистром $R2$. Если $R2 < R1$, то копируются регистры от $R1$ до последнего, затем копируется регистр $R(0)$ и далее, кончая регистром $R2$. Вторая действует аналогично в обратном направлении — содержимое ячейки G загружает в регистр $R1$, затем — $G+1$ в $R1+1$ и т. д.

Сохранение регистров в стеке

В машине вводится пара команд — копирования группы регистров в стек и загрузки группы регистров из стека. Они располагаются в подпрограмме так же, как и на рис. 5.14. Только теперь не нужно резервировать ячейки, начиная с G.

Если подпрограмме для промежуточных переменных не хватает регистров и подпрограмма использует также ячейки памяти в качестве рабочих, то их отводят тоже в стеке.

Принято называть стековым фреймом (frame — рамка, кадр) часть пространства стека, используемого подпрограммой (рис. 5.15).

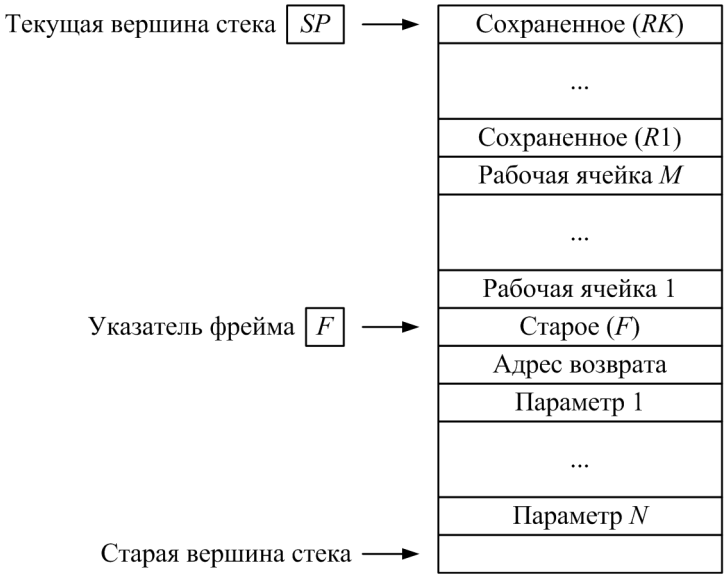


Рис. 5.15. Сохранение регистров в стеке

На рисунке изображен регистр F — указатель фрейма. Это один из регистров, обычно один и тот же для данной машины. Он используется для указания некой средней точки фрейма: ниже нее — параметры (первый — в ячейке $(F)+2$, второй — в $(F)+3$ и т. д.), выше — рабочие ячейки (в $(F)-1$, $(F)-2$ и т. д.).

Работа со стеком происходит в следующем порядке.

Главная программа:

1. Заносит в стек N параметров, необходимых подпрограмме.
2. Передает управление подпрограмме, занося в стек адрес возврата.

Подпрограмма:

1. Сохраняет в стеке старое содержимое регистра F.
2. Копирует в F содержимое указателя стека SP.

3. Вычитает из указателя стека число M — количество используемых ею рабочих ячеек.
4. Сохраняет в стеке старые значения тех K регистров, которые она будет использовать.
5. Выполняет свою основную функцию, используя K регистров, M рабочих ячеек и пространство стека выше текущей вершины.
6. Восстанавливает из стека старые значения регистров с $R1$ по RK.
7. Восстанавливает старое значение регистра F.
8. Возвращает управление главной программе, используя сохраненный адрес возврата.

Главная программа:

1. Выталкивает из стека N параметров.
2. Продолжает работу, используя значения переменных, переданные ей из подпрограммы (часто эти значения помещаются подпрограммой в стек на место параметров).

Несколько замечаний.

Первое — термин "выталкивание" означает всего-навсего перемещение указателя стека вниз. "Вытолкнуть N параметров" — это увеличить на N содержимое SP. При этом параметры в ячейках памяти, бывших до "выталкивания" в стеке, сохраняются.

Второе — использование указателя фрейма удобно, но не обязательно. Конечно, адреса параметров и рабочих ячеек можно отсчитывать от указателя стека.

Третье — на рис. 5.15 изображен самый полный вариант фрейма. В частных случаях может не быть специального поля рабочих ячеек — подпрограмма пользуется для промежуточных результатов только стеком или только регистрами, параметры могут передаваться через регистры.

Четвертое — в некоторых машинах имеется команда возврата вида $\text{БП}_{SP}\ N$. Команда выполняется так:

1. ((УК)) $\rightarrow$ РК.
2. (УК) + 1 $\rightarrow$ УК.
3. Дешифрировать КОП.
4. $(SP)+ \rightarrow$ УК.
5. $(SP) + N \rightarrow SP$.
6. Перейти к пункту 1.

То есть выталкивание параметров из стека выполняется в подпрограмме, командой возврата.

Использование стекового фрейма, как и вообще стека, очень удобно при написании программы, но за это удобство приходится расплачиваться некоторым снижением эффективности программы — увеличением времени исполнения и, часто, увеличением занимаемой памяти. Поэтому, в зависимости от обстоятельств, используются все перечисленные приемы передачи параметров, сохранения регистров и адреса возврата и их различные комбинации.

5.1.5. Настройка по месту

Использование подпрограмм делает актуальной новую проблему. Теперь оказывается, что программа в целом — набор взаимосвязанных блоков — главная программа и подпрограммы. Блоки эти пишутся независимо, связаны лишь минимальными формальными требованиями — соглашениями о подпрограммах. И конечно, они не могут быть ориентированы на согласованное расположение в памяти. Подпрограмм много, комбинаций, в которых они могут быть использованы, еще больше. Длина главной программы при составлении подпрограмм неизвестна. Короче, подпрограммы оказываются в памяти вовсе не на том месте, на которое они рассчитаны. А это приводит к тому, что некоторые адреса оказываются в подпрограммах неверными.

Обычно подпрограммы пишутся так, чтобы они могли правильно работать, если их расположить в памяти с ячейки № 0. Выделим два типа команд в подпрограмме: команды, использующие константы, размещенные в конце подпрограммы, и команды, передающие управление внутри подпрограммы. На рис. 5.16 изображена схема подпрограммы с двумя такими командами — в ячейках β и γ.

Если эту подпрограмму разместить в памяти, например, с ячейки 100, то для правильной работы надо изменить в ее командах некоторые адреса. Иначе команда, находившаяся в ячейке β, а теперь попавшая в ячейку $\beta+100$, будет брать константу из ячейки 40, а w переместилась в ячейку 140; команда, бывшая в ячейке γ (теперь — в $\gamma+100$) — передавать управление в ячейку 10 вместо 110.

Адреса команд и констант, принадлежащие телу подпрограммы, называются *внутренними*.

Команды подпрограммы, использующие внутренние адреса, должны быть модифицированы в соответствии с фактическим местом, занимаемым подпрограммой. Используется термин "настройка по месту".

Настройка по месту — простая формальная процедура. Надо ко всем внутренним адресам подпрограммы прибавить адрес начала подпрограммы. Для

облегчения настройки по месту в современных машинах вводится особый способ адресации — адресация с базированием. Это относится только к прямым адресам памяти, т. к. только они и могут быть внутренними.

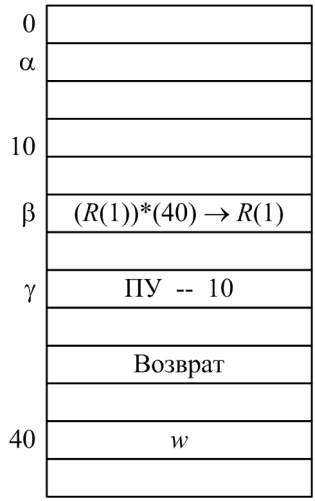


Рис. 5.16. Внутренние адреса подпрограммы

Команды с базированием типа "регистр — память" и "память — память" выглядят соответственно так: $P\ \ R1\ \ B2D2$ и $P\ \ B1D1\ \ B2D2$.

Первый адрес первой команды — регистр $R1$, первый адрес второй команды — $E1=(B1)+D1$. Второй адрес у обеих команд одинаковый — $E2=(B2)+D2$. $B1$ и $B2$ — номера базовых регистров. В машинах с регистрами общего назначения в качестве базовых можно использовать любые регистры. $D1$ и $D2$ — положительные числа, называемые смещениями. С базированием могут быть как команды арифметико-логического типа, так и команды передачи управления.

Используем в качестве базового регистра, например, $R(6)$.

В начале подпрограммы, например в ячейке α, следом за командами сохранения регистров выполним команду перехода к подпрограмме формата "регистр — регистр" вида: ПП $R(6)\ \ 0$.

Она передаст управление следующей команде — в следующую ячейку и зашлет адрес возврата — $100+\alpha+1$ в регистр $R(6)$.

В ячейках β и γ используем команды формата "регистр — базируемая память": $P\ \ R(1)\ \ R(6)\ \ 40-\alpha-1$ и ПУ $\sim\ R(6)\ \ 10-\alpha-1$.

Теперь они будут работать правильно. $40-\alpha-1$ — смещение константы w относительно ячейки $\alpha+1$, а в $R(6)$ реальный адрес команды, которая прежде была в ячейке $\alpha+1$. То же и со вторым адресом.

Замечания

- Объем настройки по месту можно уменьшить, применяя для констант непосредственную адресацию (если она есть в системе команд машины). Непосредственный адрес нигде не упоминается явно и его не надо настраивать. Если в системе команд имеются команды относительной адресации, то надо использовать их (разумеется, в случае, если адрес расположен достаточно близко) — они тоже не требуют настройки.
- Базирование может использоваться вместе с индексированием. То есть обычно в системе команд имеется формат "регистр — индексируемая базированная память", команды вида $P\ R1\ I2B2D2$, у которых второй исполнительный адрес вычисляется так: $E=(I1)+(B1)+D1$.
- Для окончательного связывания главной программы с подпрограммами остается в главной программе правильно назвать адреса переходов к подпрограмме. Если подпрограммы размещаются в памяти программистом, то их начальные адреса программисту известны. Однако обычно подпрограммы вызываются из библиотеки и размещаются в памяти той или иной разновидностью специальных программ. Они же заменяют условный адрес перехода к подпрограмме на реальный. Но это — вне рамок нашей темы.

5.2. Операции ввода/вывода

С циклами и подпрограммами тесно связаны операции ввода/вывода. Имеются три технологии выполнения операций ввода/вывода:

- ❒ программно управляемый ввод/вывод;
- ❒ ввод/вывод по прерываниям;
- ❒ прямой доступ к памяти.

5.2.1. Программно управляемый ввод/вывод

Взаимодействие программы с внешним устройством при программно управляемом выводе происходит примерно по такой схеме:

1. Команды программы выдают адрес устройства и вид работ (OUT).
2. Следующая команда запрашивает готовность устройства (читает содержимое регистра состояния); устройство работает медленно и может быть не готово просто потому, что не успело выполнить чтение (запись) предыдущего данного; если устройство не готово, программа ждет (выполняет так называемый "цикл ожидания").

3. Команда программы требует записать (вывести) слово, сообщая, если необходимо, из какой ячейки памяти требуется читать (в некоторых компьютерах чтение из памяти может производиться только из одного, фиксированного регистра, в таком случае его указывать не надо, но предварительно выводимое данное должно быть занесено в этот регистр) и в какое место внешнего устройства направить.
4. Контроллер принимает слово из памяти в свой регистр данных, устанавливает в 0 признак готовности (сигнал "занято"), а затем передает слово в указанное место внешнего устройства; по окончании работы контроллер заносит в свой регистр состояния единицу — признак готовности.
5. Во все время работы контроллера (сразу после передачи ему данного) программа выполняет цикл ожидания: ждет окончания работы контроллера — появления сигнала "свободно"; затем программа продолжает работу.

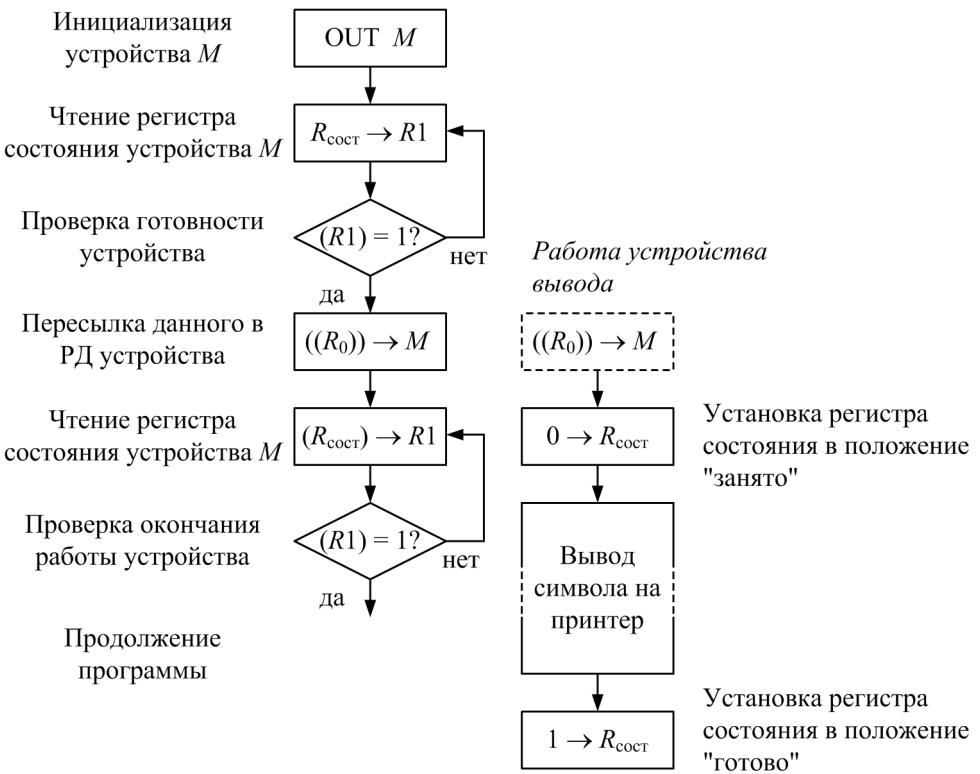


Рис. 5.17. Взаимодействие центрального процессора с устройством вывода на печать

На рис. 5.17 в виде двух параллельных блок-схем изображено взаимодействие центрального процессора с устройством вывода на печать. На левой блок-схеме — два цикла ожидания.

Последовательность работы центрального процессора и контроллера изображена на рис. 5.18.

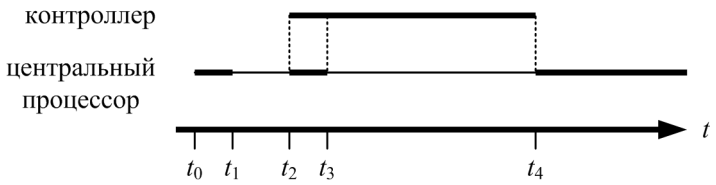


Рис. 5.18

На отрезках $t_1 \div t_2$ и $t_3 \div t_4$ центральный процессор выполняет циклы ожидания. На $t_2 \div t_3$ — передает символ контроллеру.

Аналогично центральный процессор взаимодействует с контроллером при вводе.

Главное достоинство программно управляемого ввода/вывода — его простота.

5.2.2. Ввод/вывод по прерываниям

При программно управляемом вводе/выводе процессор во время работы контроллера не может заняться чем-то полезным: он должен отслеживать появление в регистре контроллера сигнала готовности. Механизм прерываний позволяет изменить ситуацию.

Прерывания

Иногда результат арифметической операции не может поместиться в отведенной для него ячейке. Такой случай называют переполнением. Возможно также возникновение ситуаций, не являющихся, строго говоря, переполнением, которые, тем не менее, делают невозможным продолжение вычислений. Это — ноль в знаменателе, отрицательный аргумент при извлечении квадратного корня. Сюда же можно отнести ошибочно оказавшиеся в команде несуществующие коды операций и несуществующие адреса. Первые вычислительные машины при возникновении таких ситуаций останавливались, и на пульте управления машиной загорался специальный индикатор. С точки зрения достижения максимальной эффективности компьютеру не следует задерживаться на этой ситуации, желательно отреагировать на нее быстро, автоматически. При возникновении аварийной ситуации следует немедленно

снять выполняемую программу и запустить новую, предварительно зафиксировав обстоятельства, в которых произошло прерывание (это поможет программисту при поиске ошибки в программе), или как-то иначе отреагировать на возникшую нестандартную ситуацию. Быстрее всего это можно сделать с помощью специальных программ — обработчиков прерываний. Понятно, что каждый тип прерываний, например деление на ноль или несуществующий код операции, должен обрабатываться по-разному.

Типы прерываний и соответствующие обработчики пронумерованы. Они постоянно хранятся в основной памяти, составляя библиотеку, аналогичную библиотеке подпрограмм. Каталог этой библиотеки называется *таблицей векторов прерываний*. Вектор прерывания — последовательность нескольких (N) байтов. Их число постоянно для каждой машины. Оно определяется количеством байтов, которое нужно для задания физического адреса памяти (для процессоров Intel, например, $N=16$). Место каждого вектора в таблице определяется его номером. Вектор прерывания с номером M хранится в ячейке $A_T+(M-1)\times N$, где A_T — адрес начала таблицы.

Когда при выполнении программы процессор обнаруживает, что не может выполнить очередную команду, он определяет тип прерывания, т. е. его номер. По номеру формирует адрес соответствующего вектора, по вектору формирует адрес обработчика и передает ему управление, предварительно запомнив адрес команды, вызвавшей прерывание и регистр флагов. После обработки прерывания (например, выдачи соответствующего сообщения) обработчик, следуя своей логике, может прекратить выполнение программы или вернуться к ее выполнению со следующей команды, восстановив для этого указатель команд и регистр флагов.

Обработчики прерываний можно считать разновидностью подпрограмм. Отличие заключается только в технике взаимодействия главной программы с подпрограммами и с обработчиками прерываний. Во всех перечисленных случаях в главной программе нет команд обращения к обработчикам, прерывание, обычно, возникает внезапно.

Реализуется такая последовательность событий:

1. Главная программа вызывает прерывание.
2. Центральный процессор сохраняет указатель команд и регистр признаков, устанавливает новое значение указателя команд.
3. Обработчик прерывания сохраняет все регистры (или только те, которые он использует), выполняет обработку прерывания, восстанавливает старые значения регистров, вызывает выход из прерывания.
4. Центральный процессор восстанавливает старое значение указателя команд и регистра признаков.
5. Главная программа продолжает работу.

Обработчик прерывания и контроллер

В современных машинах обращение к внешнему устройству из главной программы тоже вызывает прерывание. Это продиктовано необходимостью упростить для пользователя и стандартизовать программирование операций ввода/вывода, обеспечить независимость программирования ввода/вывода от особенностей того или иного периферийного устройства. Кроме того, большинство операций ввода/вывода — групповые, т. е. на диск и с диска записывается (читается) последовательно сразу большая группа кодов, на принтер выводится последовательно сразу целая строка и т. д. Это значит, что надо организовывать циклическую передачу данных. Организацию цикла также можно перенести в обработчик прерывания.

По окончании работы контроллер выдает сигнал прерывания. Это позволяет центральному процессору вместо цикла ожидания выполнять какую-нибудь работу.

Здесь можно привести такое сравнение. Предположим, преподаватель дал учащимся контрольную работу и ждет результатов. Возможны два варианта ожидания. Первый: преподаватель предлагает закончившему поднять руку, а сам сидит и смотрит, когда рука поднимется. Второй: преподаватель предлагает закончившему громко заявить: "Я закончил", а сам принимается читать книгу. Получив сигнал готовности, преподаватель кладет в книгу закладку и принимает контрольную.

Рассмотрим для примера работу программы, которой по ходу вычислений потребовалось вывести на принтер массив данных. Буфер принтера может хранить только одну строку, а массив состоит из нескольких строк.

На рис. 5.19 изображен пример последовательности действий главной программы, обработчика прерываний и контроллера.

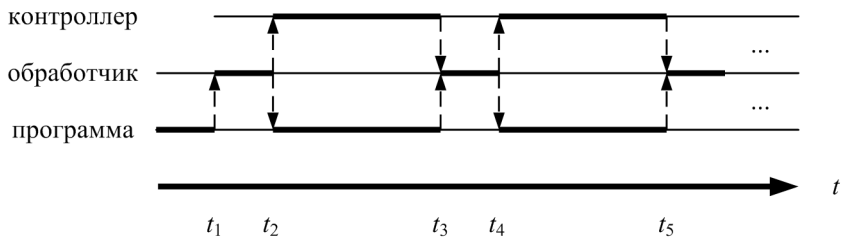


Рис. 5.19

В момент t_1 программа обращается к принтеру. Происходит прерывание, начинает работать обработчик прерывания.

В промежутке $t_1 \div t_2$ обработчик копирует первую строку массива в буфер принтера и дает контроллеру команду начать печать. Одновременно дается команда выхода из прерывания, восстанавливается контекст (значения регистров) главной программы.

В промежутке $t_2 \div t_3$ принтер печатает строку и, одновременно, центральный процессор продолжает выполнять главную программу с того места, где она была прервана.

В момент t_3 заканчивается печать строки и контроллер выдает сигнал прерывания. Центральный процессор заканчивает выполнение очередной команды главной программы, сохраняет ее контекст и вызывает обработчик.

Далее на протяжении времени $t_3 \div t_5$ повторяется последовательность действий с момента t_1 до t_3. Эти повторения происходят и далее, пока не будет исчерпан массив.

5.2.3. Прямой доступ к памяти

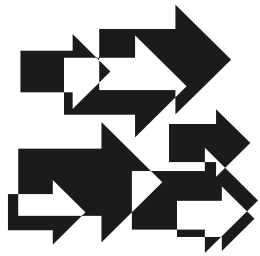
Использование механизма прерываний позволило часть рутинной работы перенести из основной программы в подпрограмму — обработчик прерываний. Это, во-первых, облегчает труд программиста, автоматизируя часть его работы, во-вторых, делает возможным экономить процессорное время за счет одновременного с вводом/выводом выполнения программы.

И все же потери времени на ввод/вывод остаются. Во-первых, при работе обработчика прерываний процессор не стоит, но и не выполняет основную программу. Во-вторых, передача всех данных происходит через центральный процессор: чтобы переслать слово с внешнего устройства в оперативную память (или обратно), необходимо в РАД указать адрес в памяти, а само данное сначала поместить в РДП, а уже затем отправить в память (или обратно).

При прямом доступе к памяти (ПДП) используется специальный контроллер, который берет на себя все функции обмена. В отличие от механизма прерываний он не требует сохранения и восстановления регистров, т. к. использует собственные регистры. Организацию цикла контроллер ПДП также берет на себя. Он не использует регистры РАД и РДП центрального процессора, фактически дублируя их.

Единственная ситуация, в которой все же происходят потери времени — одновременное обращение к памяти центрального процессора и контроллера ПДП. Так как внешнее устройство обычно не желательно задерживать (например, при вводе — вводимые данные нужны как можно скорее), приоритет в таком конфликте имеет контроллер ПДП. В этот момент центральный процессор простаивает. Но и здесь простой случается не всегда — процессор может выполнять операцию типа "регистр — регистр" одновременно с обращением к памяти контроллера.

ГЛАВА 6



Параллельность работы и иерархия памяти

6.1. Основные идеи

Производительность компьютера в значительной степени зависит от главных характеристик памяти — быстродействия и емкости. Существует большое количество различных физических и технических принципов устройства памяти. С потребительской точки зрения они различаются в основном быстродействием и стоимостью единицы памяти. Естественно, чем быстрее устройство, тем оно дороже. Использование наиболее быстродействующей памяти в объеме, достаточном для обеспечения высокой производительности компьютера, сделало бы его чрезвычайно дорогим. Во всяком случае, массовое использование такого компьютера было бы исключено. К счастью, есть две идеи, позволяющие строить компьютеры с памятью большой емкости, по быстродействию близкой к самым дорогим образцам и с умеренной стоимостью. Первая — построение памяти с иерархической структурой, вторая, дополняющая первую, — исключение простоев центрального процессора за счет организации параллельных действий различных устройств.

6.1.1. Иерархия памяти. Идея

На рис. 6.1 — условное изображение пирамиды памяти компьютера (масштаб не выдержан — на деле размеры памяти снизу вверх убывают гораздо быстрее).

Наверху пирамиды находятся регистры — самая быстрая память. Компьютеры различаются по количеству регистров, но в любом случае их число измеряется единицами или десятками. В регистрах располагаются наиболее часто используемые данные. Команды в регистрах располагаться не могут. Иногда эти регистры называют числовыми или арифметическими регистрами, чтобы

подчеркнуть их отличие от внутренних регистров процессора, недоступных программисту.

Рис. 6.1. Иерархия памяти

Размеры кэш-памяти — от нескольких килобайт до нескольких мегабайт. Кэш-память — скрытая память. Она используется в качестве служебной самим компьютером, как правило, без участия программиста, от которого она и скрыта. Там, как и в основной памяти, могут располагаться и данные, и команды. Именно здесь часто нарушается фон-неймановский принцип архитектуры — обычно на первом уровне используются раздельные кэши для данных и для команд. Во многих машинах используется только два уровня кэш-памяти.

Ниже идет основная память (другие названия: "первичная память", "оперативная память"). В персональных компьютерах объем основной памяти сегодня чаще всего 1—2 Гбайт.

В основании пирамиды — вторичная память. Ее называют также массовой или внешней памятью. Обычно это — дисковая память. Команды и данные, находящиеся во вторичной памяти, могут быть использованы процессором только после того, как они будут скопированы в основную память. Объем дисковой памяти в современном персональном компьютере обычно — 80—120 Гбайт. Эти цифры постоянно и очень быстро растут.

Вообще, чем выше уровень памяти, тем выше стоимость каждого ее байта, тем она быстрее, надежнее и тем ее меньше.

Эффективность иерархической организации памяти связана с пространственной и временно́й локальностью данных и команд (используется также выражение "локальность ссылок").

Пространственная локальность данных означает, что данное, которое сейчас выбирается из памяти, находится недалеко от того места, где хранилось дан-

ное, выбранное непосредственно перед этим. То же относится и к командам. Временна́я локальность означает, что пространственная локальность сохраняется достаточно долго. Как понимать "недалеко" и "достаточно долго" — зависит от вида памяти. Скажем, программа оказывается полностью локальной для оперативной памяти, если она целиком в ней помещается. Для регистров данные полностью локальны, если они целиком помещаются в регистрах. Точное определение и мера локальности выходят за пределы нашей темы.

Заметим, что любая программа обладает значительной степенью локальности, так сказать, стихийно. Команды выполняются, как правило, подряд, имеется большое число небольших циклов, на которых программа задерживается на определенное время. Данные располагаются в памяти группами, относящимися к определенным участкам программы. Степень локальности можно усилить целенаправленным программированием.

Проиллюстрируем идею использования иерархической памяти.

Пусть на некотором уровне памяти находится фрагмент программы, состоящий из последовательности примерно равных по длине участков. Пусть участки попеременно, в любом порядке, передаются на более высокий уровень памяти для исполнения (рис. 6.2).

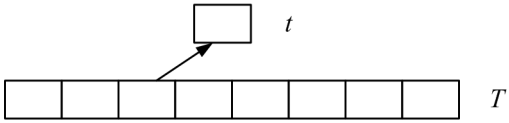


Рис. 6.2

Большинство участков содержат циклы. Это значит, что многие команды и данные используются в этих участках по нескольку раз.

Пусть в среднем каждый байт при выполнении программы вызывается в процессор N раз. Пусть время выборки из памяти нижнего уровня — T, а из верхнего — t. Тогда среднее время выборки одного байта из памяти, скомбинированной из двух уровней:

$$\tau = t + \frac{T}{N}.$$

При достаточно большом N τ оказывается близким к t. То есть практически все выглядит так, будто мы располагаем памятью с объемом, равным объему памяти нижнего уровня, и быстродействием, чуть меньшим быстродействия памяти верхнего уровня.

6.1.2. Параллельность работы. Идея

Параллельная работа разных уровней памяти, а также устройств ввода/вывода обеспечивается комплексом программных и аппаратных средств. Совокупность программных средств является частью операционной системы (ОС) — программы, постоянно находящейся в памяти компьютера; аппаратные средства — контроллеры и буферы, а также механизм прерываний, "зашитый" в схему процессора.

Буфер — это запоминающее устройство сравнительно небольшой емкости, предназначенное для кратковременного хранения информации. Обычно буфер работает по принципу FIFO (First In — First Out, первым пришел — первым вышел). Буфер по очереди принимает от памяти одного уровня некоторые единицы информации (байты, слова) и по очереди передает их в память другого уровня.

Контроллер — это специализированный процессор, способный управлять одним из устройств компьютера. В частности память на каждом уровне обычно управляется своим контроллером. Именно он управляет процессом передачи информации из буфера в память и из памяти в буфер. Буфер, расположенный между двумя уровнями памяти, должен работать со скоростью памяти более высокого уровня.

Взаимодействие двух соседних уровней памяти происходит в два этапа. Если требуется передать информацию с верхнего уровня на нижний, то на первом этапе контроллер памяти верхнего уровня копирует информацию в буфер. Запись происходит быстро — со скоростью работы памяти верхнего уровня. Затем, на втором этапе, память верхнего уровня и ее контроллер освобождаются и работают дальше, а контроллер памяти нижнего уровня управляет копированием содержимого буфера в память нижнего уровня. В идеале получается, что память верхнего уровня всегда работает с максимально возможной для нее скоростью, не дожидаясь, пока закончит прием информации более медленная память нижнего уровня. Правда, совсем избежать затрат, связанных с необходимостью размещения части информации в памяти нижнего уровня, не удается даже в идеале — затрачивается время на пересылку в буфер (первый этап). Но это время мало.

Аналогично, в два этапа, происходит загрузка памяти верхнего уровня из памяти нижнего уровня через тот же буфер. Однако в этом случае дело осложняется тем, что запрошенная из памяти нижнего уровня информация обычно требуется немедленно, для дальнейшего выполнения программы, и пока ее нет, верхний уровень памяти простаивает. Чтобы предотвратить эти простои применяют упреждающую выборку — чтение той информации, которая может понадобиться вскоре, но не немедленно.

В некоторых случаях упреждающую выборку может выполнять контроллер по своей инициативе. Если после чтения фрагмента из памяти нижнего уровня в буфер в нем остается свободное место, то контроллер продолжает чтение дальше. При записи в память верхнего уровня, если после затребованного фрагмента имеется свободное пространство, в него производится запись следующего фрагмента. Исходя из принципа локальности, велика вероятность, что эта информация вскоре понадобится.

Инициатором упреждающей выборки может быть также программа. Программист или транслятор может воспользоваться командами чтения информации из памяти нижнего уровня в верхний так, чтобы они выполнялись до того, как выбираемая информация понадобится. Тогда выборка будет выполняться одновременно с выполнением программы.

В общем, если все складывается удачно и простои памяти верхнего уровня удается предотвратить, время выборки τ из памяти, скомбинированной из двух соседних уровней, будет определяться формулой

$$\tau = t + \frac{t}{N}.$$

Таким образом, в идеале τ незначительно больше t и уж во всяком случае $t < \tau \le 2t$.

Однако далеко не всегда удается организовать упреждающую выборку и главным средством минимизации простоев процессора является эксплуатация компьютера в многопрограммном режиме, в котором центральный процессор выполняет одну из программ, а контроллеры, одновременно с этим, выполняют операции ввода/вывода или обмена.

6.1.3. Технология взаимодействия уровней памяти

На самом деле все обстоит не так удачно.

"Промахи" при обращении со стороны процессора к памяти верхнего уровня (отсутствие в ней нужного фрагмента) далеко не всегда удается предотвратить упреждающей выборкой. Тогда приходится ждать подкачки нужного фрагмента из нижнего уровня.

Чтобы уменьшить количество промахов, целесообразно копировать на верхний уровень несколько фрагментов программы и данных. Например, это может быть участок основной программы, подпрограмма, к которой имеется обращение на данном участке, два-три массива данных, стек.

Постоянно анализировать программу с целью определения участков, которые целесообразно в каждый момент времени держать в памяти верхнего уровня, было бы очень сложно. Поэтому действуют два формальных правила.

Первое: память на всех уровнях делится на отдельные блоки; все блоки на каждом уровне одинаковой длины; блок на более высоком уровне меньше или равен блоку нижнего уровня.

Второе: если произошел промах по чтению, в блок памяти верхнего уровня копируется фрагмент, содержащий затребованный адрес.

Так как объем памяти верхнего уровня значительно меньше, чем нижнего, в верхнюю память можно скопировать только часть нижней памяти. Для того чтобы имелась возможность установить, какие именно фрагменты нижней памяти скопированы в верхнюю и в какие ее блоки, в памяти верхнего уровня хранится каталог. В каталоге каждому блоку верхней памяти сопоставляется некоторый описатель. Описатели блоков оперативной памяти называют дескрипторами, блоков кэш-памяти — тегами.

Тег/дескриптор — это строка, содержащая указание о том, из какого места памяти нижнего уровня скопирован содержащийся в блоке фрагмент, а также поле признаков.

Поле признаков содержит подполя:

- R — признак используемости; значение R показывает, давно ли в последний раз использовался данный фрагмент или как часто он используется;
- V — признак действительности; если значение признака 0, то фрагмент не соответствует никакому блоку;
- M — признак модификации; если значение признака 1, то фрагмент "грязный" — в него производилась запись и он, возможно, уже не является копией соответствующего фрагмента нижнего уровня.

Признак используемости R (обычно состоит из 2—3 бит, может отсутствовать) позволяет определить фрагмент, который следует первым вытеснить из памяти данного уровня. Существуют различные принципы формирования R.

Самый распространенный — замещение наиболее давно использовавшегося (Least Recently Used, LRU). Он реализуется аппаратно, например, так. Через определенные промежутки времени R всех тегов, не достигшие максимального значения, увеличиваются на единицу. При каждом обращении к фрагменту его признак R обнуляется. При необходимости освободить место для записи нового фрагмента удаляется один из фрагментов с максимальным значением R.

Еще один принцип — замена наиболее редко использующегося фрагмента (Least Frequently Used, LFU). В этом случае R — счетчик обращений. При каждом обращении к фрагменту R увеличивается на единицу. Удаляется фрагмент, в теге которого R имеет наименьшее значение.

Можно вести отдельный список фрагментов. При обращении к фрагменту соответствующий элемент списка переставляется в конец. Удаляется первый элемент.

В любом варианте приходится выполнять дополнительную работу: просматривать каталог в поисках строки с минимальным или максимальным значением R, или вести дополнительный список. Поэтому существует и такой принцип: заменять случайно выбранный фрагмент. Ясно, что в этом случае признак используемости не нужен.

Недействительный фрагмент может возникнуть, например, при удалении программы. Тогда следует очистить занимаемые этой программой блоки ОП и кэш. В таком случае не производится физическое стирание фрагментов, просто они объявляются свободными. Для этого достаточно установить для них признаки недействительности. Другой случай: когда производится запись с диска в ОП, может оказаться, что некоторые блоки, в которые производится запись, представлены в кэше. ОС отслеживает такие ситуации и помечает в кэше соответствующие блоки как недействительные.

При записи процессором результатов операций в память верхнего уровня в ней возникает модифицированный блок, отличающийся от фрагмента, с которого он был скопирован. Для того чтобы различать модифицированные и немодифицированные блоки, в тегах и дескрипторах вводится признак модифицированности (модификации). Признак модификации (используется не всегда) устанавливается, когда процессором выполняется запись в блок. Признак указывает, что фрагмент, хранящийся в этом блоке, "грязный", т. е. перестал быть точной копией фрагмента, хранящегося в памяти более низкого уровня. Это не существенно, пока фрагмент регулярно используется. Как только принимается решение о замене фрагмента, прежде чем записывать в блок новый фрагмент, надо сохранить модифицированный фрагмент в блоке памяти нижнего уровня. При $M = 0$ сохранение фрагмента (выгрузка) не производится.

Вопрос о количестве и размерах фрагментов на разных уровнях памяти очень трудный.

С одной стороны, время пересылки n байт с одного уровня памяти на другой включает в себя некоторую настроечную работу, не зависящую от длины фрагмента (поиск по каталогу места для фрагмента, запись в каталог новой информации и др.). При этом время, приходящееся на один байт, можно выразить формулой

$$t = t_0 + t_1/n,$$

где t_0 — время пересылки одного байта, а t_1 — время настройки. Следовательно, чем длиннее фрагмент, тем лучше.

С другой стороны, в длинном фрагменте может оказаться много данных или команд, которые за время интенсивного использования этого фрагмента ни разу не будут востребованы. И чем больше фрагмент, тем число таких элементов больше. В то же время программе будут требоваться данные и команды из других фрагментов, что приведет к частой их смене.

Основными характеристиками каждого уровня памяти с точки зрения общего быстродействия являются время пересылки слова (или байта) на верхний уровень и частота промахов.

Рассмотрим совместную работу двух соседних уровней. Пусть время обработки (пересылки наверх) одного слова на верхнем уровне t, на нижнем — T. Пусть при обращении к памяти верхнего уровня один промах приходится в среднем на N обращений. Для идеально сбалансированной памяти должно выполняться соотношение

$$T \le t \times N.$$

Тогда правильно организованная упреждающая выборка будет гарантировать работу памяти из двух уровней со скоростью памяти верхнего уровня. Влияние того, что N — среднее число, а реально будут промахи и чаще и реже, демпфируется использованием промежуточного буфера.

К сожалению, не всегда удается правильно предугадать, какие фрагменты памяти нижнего уровня потребуются в ближайшее время на верхнем уровне. В результате вызываются ненужные фрагменты, а нужных вовремя не оказывается на верхнем уровне.

6.2. Виртуальная память

Работа в многопрограммном режиме, повышая производительность компьютера, ставит новые проблемы.

Во-первых, защита программ. Необходимо сделать так, чтобы ни по ошибке, ни по злому умыслу программы не могли вмешиваться в работу друг друга.

Во-вторых, независимость адресных пространств. Использование программой памяти не должно зависеть от того, какие программы расположены в памяти вместе с ней. Каждая программа должна использовать основную память так, будто других программ в памяти нет.

В-третьих, чтобы размещать в основной памяти одновременно несколько программ, памяти должно быть много.

Все эти проблемы решает виртуальная память. Рассмотрим совместную работу основной и вторичной памяти. Если программа полностью, вместе со всеми данными, помещается в основной памяти, то медленной дисковой памяти

для нее как бы не существует. Если же программа велика и в основной памяти полностью не помещается, то используется *виртуальная память*. Это воображаемая память, по объему равная максимально адресуемой памяти (например, 4 Гбайт). Главная идея виртуальной памяти — использовать внешнюю (дисковую) память как продолжение основной, причем организовать все на аппаратном уровне или на уровне ОС так, чтобы программист не замечал границы между основной и внешней памятью.

6.2.1. Диск

Виртуальная память построена на тесном взаимодействии оперативной и дисковой памяти, поэтому уместно рассмотреть принципы их работы.

Основную память называют *памятью прямого доступа*. При прямом доступе время доступа к любой ячейке памяти не зависит от ее номера. Существует также *память последовательного доступа*. Самый яркий ее представитель — память на магнитных лентах. Она удобна для хранения архивов и очень дешева. На ленте время доступа к информации зависит от ее физического расположения. Ячейки, расположенные прямо под считывающей головкой, доступны немедленно, но чем дальше они от головки, тем дольше надо ждать перемотки ленты.

Дисковая память занимает промежуточное положение. Иногда ее называют памятью прямого доступа, но это не точно. Время доступа к различным ячейкам диска примерно одинаково лишь в среднем.

Кратко об устройстве магнитного диска.

Магнитным диском, жестким диском, винчестером, просто *диском* называют запоминающее устройство большой емкости. Сегодня самые распространенные диски имеют емкость 120, 160 Гбайт (1 Гбайт = 2^{30} байт). Диск состоит из двух частей. Одна часть — набор из нескольких пластин в форме дисков (может быть только одна пластина), параллельно насаженных на общую ось. Электромотор вращает диски с постоянной скоростью во все время работы компьютера. Сегодня обычная скорость 7200 или 10 000 оборотов в минуту. Другая часть — набор головок чтения/записи. По одной головке на каждую поверхность пластины. Головки расположены на кронштейнах и могут передвигаться таким образом, чтобы оказываться на любом расстоянии от оси диска. Вместе с вращением пластин это обеспечивает головкам возможность находиться напротив любого участка пластины. Пластины покрыты с двух сторон ферромагнитным слоем, способным намагничиваться. Запись ведется с помощью головок. Они намагничивают отдельные участки диска (таких участков на поверхности каждой пластины многие триллионы). Намагниченность одной полярности принимается за единицу, противоположной — за

ноль. Эти же головки, воспринимая намагниченность участков пластин, читают записанную информацию. Информация записывается по концентрическим дорожкам. Внешняя дорожка имеет номер ноль, затем по направлению к центру идет дорожка номер один и т. д. Дорожки делятся на секторы, емкость каждого — 512 байт плюс служебная информация. Диск устроен так, что переслать меньше, чем один сектор невозможно.

Из сказанного видно, что для чтения информации с диска необходимо:

- переместить головку на нужную дорожку;
- подождать, пока диск повернется так, чтобы нужный сектор оказался напротив головки;
- переслать информацию.

Время выполнения первых двух действий называется *временем позиционирования*. Оно в сотни раз больше времени пересылки одного байта. Время позиционирования для разных ячеек диска различно — зависит от положения головки в момент запроса.

Очевидно, что для чтения N байт требуется время T:

$$T_N = t_{\text{позиционирования}} + N \times t_{\text{пересылки}}.$$

То есть целесообразно пересылать данные на диск и с диска большими порциями. Диск хранит большое число различных блоков информации (файлов). Файлы пишутся на диск, удаляются, в результате дисковое пространство оказывается *фрагментированным* — чередуются занятые и свободные секторы. Поэтому файл далеко не всегда размещается на подряд расположенных секторах — часто на диске не находится достаточно большого свободного фрагмента. Поэтому формула верна, если первое слагаемое трактовать как суммарное время многократного позиционирования.

Кроме того, имеется большая разница между временем внутренней пересылки (от головки к выходу с диска) и внешней пересылки (от выхода диска к оперативной памяти). Время внешней пересылки в разы меньше. Для сглаживания этой разницы в устройство диска входит буфер (несколько мегабайт). При чтении информации с диска она накапливается в буфере, а потом из буфера передается в память. Аналогично при записи на диск информация передается в два этапа: из оперативной памяти в буфер, из буфера на диск. Пересылками руководит контроллер диска, получающий указания от центрального процессора:

- адрес (имя) устройства;
- направление пересылки (чтение или запись);
- число пересылаемых байтов;

- адрес пересылки в оперативной памяти;
- адрес на диске (номер дорожки, номер сектора).

Свойство локальности ссылок делает эффективным использование еще двух приемов, позволяющих сократить время доступа к диску.

- В силу пространственной локальности после считывания некоторого сектора с высокой вероятностью могут понадобиться данные или команды, расположенные в других секторах той же дорожки. Поэтому считывается вся дорожка, на которой расположен запрошенный сектор, причем считывание начинается сразу после установки головки на нужной дорожке, без ожидания запрошенного сектора.
- В силу временной локальности запрошенные данные или команды вскоре могут понадобиться снова. Поэтому они сразу не удаляются из буфера диска, на них заводится каталог. При обращении к диску контроллер сначала проверяет, нет ли запрошенной информации в буфере, и если находит, выдает ее прямо из буфера, не считывая повторно. Такое использование буфера диска аналогично тому, как взаимодействует кэш-память с оперативной памятью. Поэтому буфер диска иногда называют *кэшем диска.*

Еще один прием повышения эффективности называется *расслоением данных*. Его применение требует наличия в компьютере M независимых дисков. Каждый файл делится на M равных частей, каждая из которых записывается на один из дисков. Запись, а потом и чтение производятся одновременно, на все диски, что значительно уменьшает время записи (чтения) файла. Его можно описать формулой

$$T_N = t_{\text{позиционирования}} + \frac{N}{M} \times t_{\text{пересылки}}.$$

Вообще-то этот прием — один из вариантов использования группы дисков и называется RAID 0. Существуют также RAID 1, 2, 3, 4, 5 и 10. Название RAID (Redundant Array of Inexpensive Discs — избыточный массив недорогих дисков) связано с тем, что во всех вариантах, кроме RAID 0 на дисках дублируется информация для повышения надежности.

Расслоение применяется и в оперативной памяти. Там оно называется *расслоением памяти*. В соответствии с этой идеей оперативная память разбивается на блоки. Это, во-первых, позволяет увеличивать память путем добавления новых блоков, а не полной замены, во-вторых, позволяет повысить быстродействие компьютера — расслоение позволяет обращаться к различным блокам памяти одновременно.

Используются два варианта расслоения. На рис. 6.3 и 6.4 изображены схемы расслоения памяти емкостью 256 Мбайт на четыре блока по 64 Мбайт. В пер-

вом случае ячейки в блоках располагаются циклически: 0000000 — в блоке 0, 0000001 — в блоке 1, 0000002 — в блоке 2, 0000003 — в блоке 3. Ячейка 0000004 — снова в блоке 0 и т. д. Во втором случае ячейки с номерами от 0000000 до 3FFFFFF расположены в блоке 0, ячейки с 4000000 по 7FFFFFF — в блоке 1 и т. д.

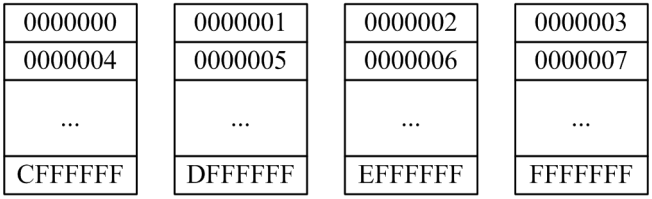


Рис. 6.3. Первый вариант использования расслоения памяти

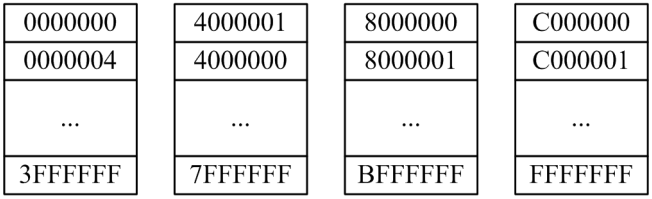


Рис. 6.4. Второй вариант использования расслоения памяти

Первый вариант целесообразен в машинах, для которых основным является однопрограммный режим. При этом, в силу локальности ссылок, велика вероятность обращения к последовательным ячейкам памяти. Это позволяет сразу после обращения к блоку k начинать считывание из следующего блока — $k+1$, не дожидаясь обращения к нему. И только последний шаг пересылки из памяти в процессор производится после подтверждения — фактического обращения к этому блоку.

Второй вариант целесообразен в машинах, для которых основным является многопрограммный режим и велика вероятность одновременного исполнения одной программы и выполнения обмена для другой программы. Тогда первая программа может обращаться к одному блоку памяти, а вторая — одновременно вести обмен с другим блоком с помощью контроллера прямого доступа.

6.2.2. Страничная организация памяти

Это один из двух способов организации виртуальной памяти.

И физическая память (реально имеющаяся в данном компьютере), и виртуальная делятся на равные, небольшие (например, 4 Кбайт, иногда — 4 Мбайт)

участки — страницы. Программа составляется так, будто в машине объем основной памяти равен виртуальной. Программа, разбитая на виртуальные страницы, первоначально располагается на диске. Затем часть начальных страниц программы, с которых начнется ее выполнение, вызывается в основную память. Операционная система ведет "таблицу страниц". В ней каждой виртуальной странице соответствует строка, в которой содержится адрес ее реального расположения в настоящий момент в основной памяти (физический адрес), ее адрес на диске (номера дорожки и сектора) и набор признаков. Строки в таблице расположены в порядке номеров страниц (номер страницы — старшие разряды ее адреса).

Программа составляется с расчетом на виртуальную память. Процессор, дешифрируя команды, вырабатывает эффективные адреса. Поскольку это вовсе не те адреса, по которым в действительности произойдет обращение, их называют *виртуальными*, а иногда *логическими адресами*. Специальный блок — контроллер управления памятью (Memory Management Unit, MMU), он же — диспетчер памяти, проверяет по таблице страниц, находится ли нужная страница в памяти. Если — да (такая ситуация называется попаданием), то преобразует виртуальный адрес в физический — реальный адрес основной памяти. Если — нет (промах, говорят также — "ошибка страницы"), то управление автоматически переходит к операционной системе, которая ищет возможность записать в ОП нужную страницу из вторичной памяти. Операционная система ведет также список свободных физических страниц. При промахе может оказаться, что свободные страницы в ОП есть. Тогда номер первой свободной страницы из списка удаляется, а сама страница используется для размещения нужной виртуальной страницы. Если список свободных страниц пуст, то отсылается обратно на диск одна из давно не использовавшихся страниц и считывается с диска на освободившееся место в ОП нужная виртуальная страница. При этом, естественно, вносятся соответствующие изменения в таблицу страниц.

На рис. 6.5 изображена таблица страниц и схема преобразования логического адреса в физический при попадании.

Здесь:

- α — номер виртуальной страницы. В таблице строк ровно столько, сколько виртуальных страниц (если виртуальная память — 4 Гбайт, а страница — 4 Кбайт, то строк в таблице — 2^{20}). Строка таблицы страниц с номером α содержит информацию о виртуальной странице с номером α. Базовый регистр таблицы страниц содержит адрес ее начала и позволяет вычислить адрес строки с номером α;
- A_α — номер физической страницы, в настоящий момент соответствующей виртуальной странице α (содержащей копию этой страницы). Физи-

ческих страниц намного меньше, чем виртуальных, и не каждой виртуальной странице соответствует в настоящий момент какая-нибудь физическая. А физические страницы в разные моменты времени могут принадлежать разным виртуальным страницам;

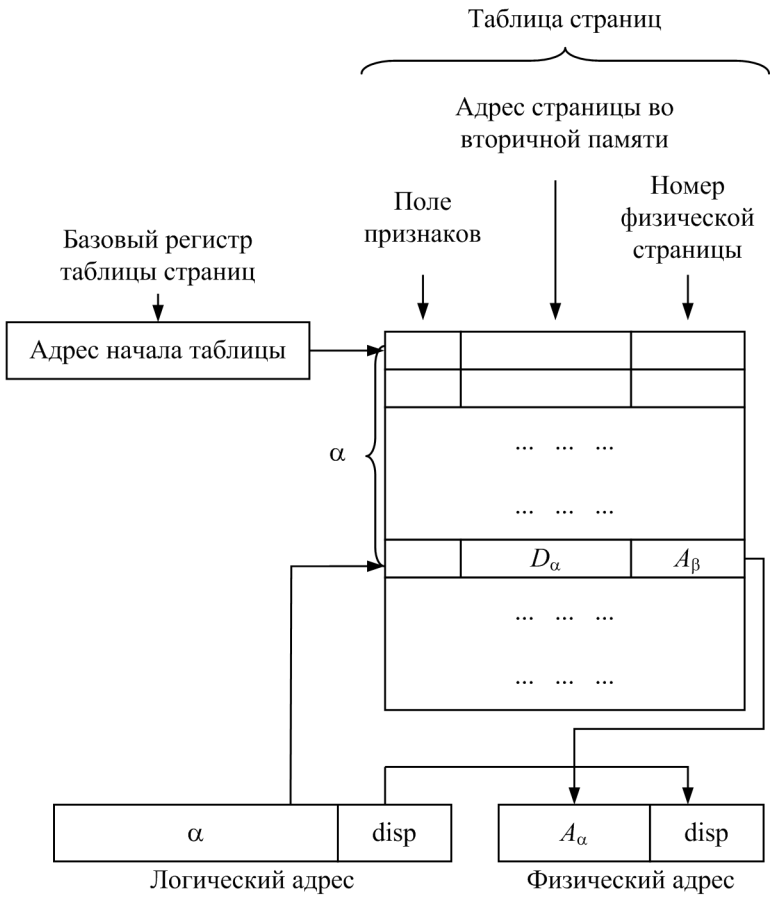


Рис. 6.5. Преобразование логического адреса в физический

- disp — смещение в байтах запрашиваемого данного или команды относительно начала страницы. Оно, естественно, одно и то же и для физической страницы, и для соответствующей виртуальной;
- D_α — адрес виртуальной страницы во вторичной памяти (на диске). Этот адрес не меняется в процессе решения задачи.

Поле признаков делится на несколько подполей:

- V_α — признак присутствия. $V_\alpha = 1$ устанавливается, если виртуальная страница α присутствует в ОП. Тогда A_α — ее номер;
- M_α — признак модифицированности. $M_\alpha = 1$ устанавливается, если в физическую страницу производится запись. Такая страница считается "гряз-

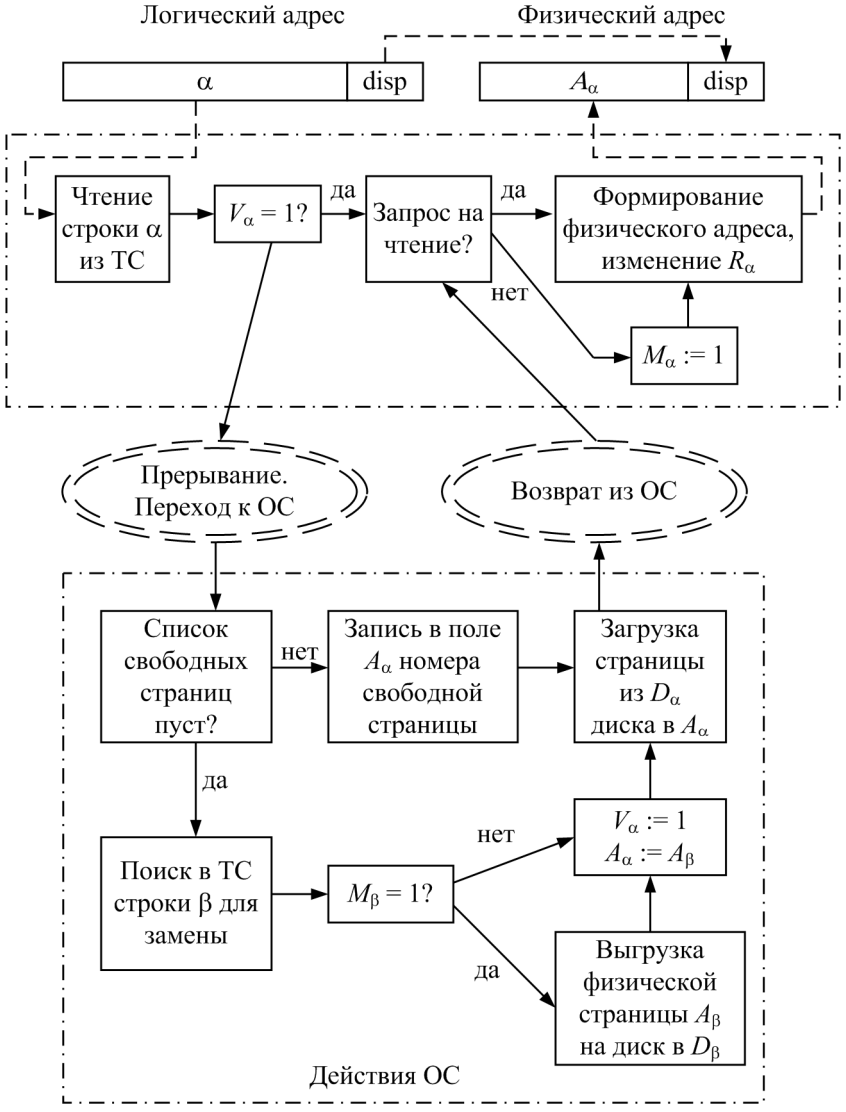


Рис. 6.6. Схема преобразования логического адреса в физический

ной". Ее содержимое не совпадает с содержимым соответствующей виртуальной страницы. Этот признак позволяет уменьшить число обращений к вторичной памяти. При $M_\alpha = 0$ заменяемая страница не пересылается на диск — там и так хранится ее копия;

- R_α — признак использования, беззнаковое целое (один или несколько битов).

Имеется также ряд других полей.

На рис. 6.6 изображена схема общего алгоритма преобразования логического адреса в физический.

В многопрограммном режиме каждой программе выделен свой набор физических страниц. Соответственно, каждая программа имеет свою таблицу страниц. Виртуальное адресное пространство у всех программ одинаковое, поэтому все таблицы страниц имеют одинаковую длину. Все они располагаются в оперативной памяти. Для переключения на выполнение новой программы достаточно занести в базовый регистр таблицы страниц (см. рис. 6.5) адрес ее таблицы страниц.

Анализ страничной организации

Таблица страниц (ТС) занимает много места в ОП (например, при 4 Гбайт виртуальной памяти и странице в 4 Кбайт ТС должна иметь 2^{20} строк, каждая около 8 байт; итого — 8 Мбайт на таблицу). Есть ряд способов уменьшения затрат памяти.

Первый — столбец с адресами страниц во вторичной памяти из ТС удаляется и переносится на диск как самостоятельная таблица — "карта диска". (Заметим, что в таком виде строки таблицы страниц совпадают с определенными ранее тегами, а сама таблица страниц — с памятью тегов. Впрочем, и в том, и в другом вариантах уместен термин "каталог".) Карта диска используется только при страничном промахе, а значит, достаточно редко. Этот способ сокращает затраты памяти примерно вдвое. Правда, дольше происходит "подкачка" — загрузка новых страниц.

Второй способ — увеличивается размер страницы. При странице в 4 Мбайт и виртуальной памяти в 4 Гбайт таблица страниц содержит только 2^{10} строк. Недостаток этого способа — неэффективное использование памяти под страницы. Часто из 4 Мбайт длительное время используется лишь незначительная часть, после чего страница выгружается, чтобы освободить место для другой страницы, на которой в ближайшее время будет использоваться тоже небольшая часть. При маленьких страницах используемые части одновременно располагаются в памяти. Очевидно, что загрузка новых больших страниц

может требоваться чаще, чем маленьких, а это вызывает также потери времени.

Третий — многоуровневая организация таблиц. На рис. 6.7 показана структура логического адреса и схема формирования физического адреса при двухуровневой организации страничной памяти.

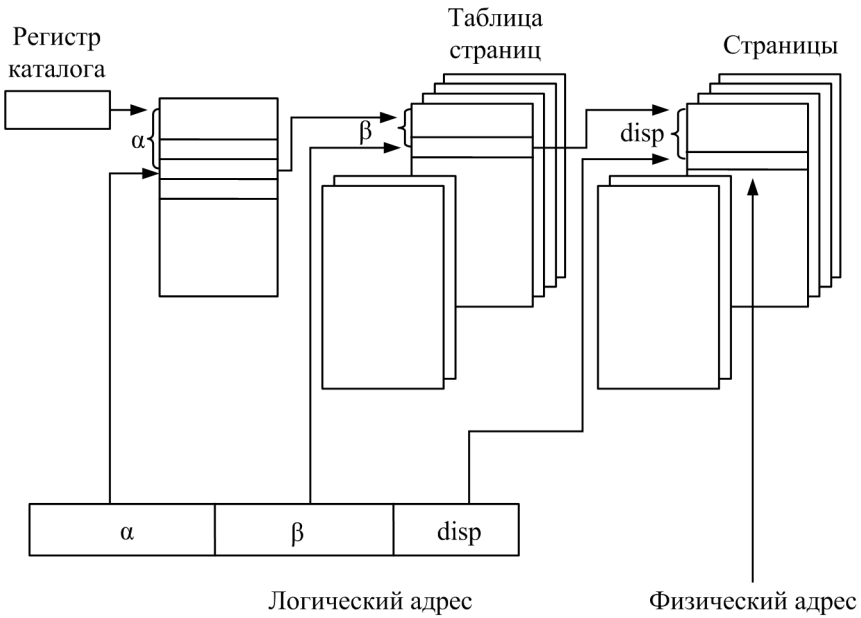


Рис. 6.7. Схема формирования физического адреса

Строки каталога таблиц и таблиц страниц изображены на рис. 6.8.

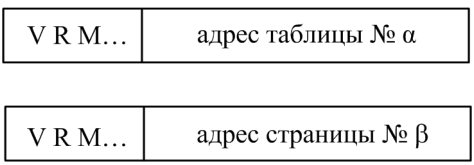


Рис. 6.8. Вид строк каталога таблиц (сверху) и таблиц страниц

При виртуальном адресном пространстве в 4 Гбайт и страницах в 4 Кбайт можно, например, использовать каталог и таблицы из 2^{10} строк. Полный набор таблиц страниц хранится на диске. В оперативной памяти для хранения таблиц выделяется место, достаточное для размещения лишь небольшой час-

ти из них. Если при обращении к памяти нужной таблицы не оказалось, она вызывается в память на место какой-либо наименее используемой или давно не используемой (применяются те же принципы, как при замене страниц).

Буфер быстрого преобразования адреса

Крупным недостатком страничной организации является необходимость при каждом запросе дважды обращаться к памяти: первый раз к таблице страниц для преобразования логического адреса в физический, второй раз — для чтения/записи по полученному адресу. Для снижения этих затрат в конструкцию процессора вводится буфер быстрого преобразования адреса (обычно используется аббревиатура TLB — Translation Lookaside Buffer). Это запоминающее устройство состоит из небольшого числа строк (обычно — от восьми до нескольких сотен). Строки TLB состоят из трех полей: поля номера виртуальной страницы, поля номера физической страницы и поля признаков.

Контроллер памяти сравнивает левую часть логического адреса (номер виртуальной страницы) одновременно с соответствующими полями сразу всех строк TLB. Если строка с таким номером находится, ее поле "№ физ. страницы" пересылается в левое поле выходного регистра, где к нему присоединяется поле disp из логического адреса, формируя тем самым физический адрес. Если нужной строки в TLB не находится, происходит обращение к таблице страниц, и соответствующая строка записывается в TLB, заменяя наименее используемую (рис. 6.9). На многих задачах число "попаданий" в TLB со-

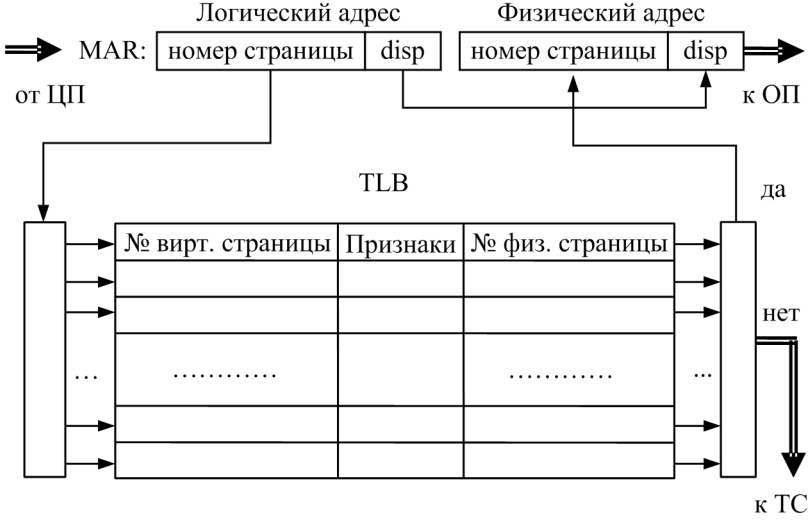


Рис. 6.9. Работа TLB

ставляет до 98%. Часто в компьютерах имеются два TLB: отдельно для данных и для команд.

6.2.3. Сегментная организация

Это второй способ организации виртуальной памяти. Его идея заключается в следующем: разделить память на сегменты — участки необходимой для программ длины и организовать работу компьютера таким образом, чтобы каждой программе был доступен только свой сегмент. На практике реализуется развитие этой идеи — каждой программе выделяется не один, а несколько сегментов. Как минимум три: сегмент кода, сегмент данных и сегмент стека. Тут действуют два соображения.

Во-первых, стремление к эффективному использованию памяти. Если ОП заполнена и требуется записать в нее новую программу, то необходимо вытеснить из памяти участок равной или большей длины, чем новая программа. Совпадение длин — редкий случай, поэтому практически при каждом таком вытеснении образуется свободный участок, недостаточный для размещения какой-либо программы. Если программы разбиты на сегменты, то средний размер сегмента меньше. Меньше и размер свободных участков. Кроме того, при загрузке одной программы можно использовать несколько участков памяти в разных местах и можно каждому сегменту подобрать участок по его размеру.

Во-вторых, при программировании происходит естественное разбиение на три вышеназванных сегмента. А если программа сложная или сложная структура данных, то и они, в свою очередь, разбиваются на части уже при программировании.

Сегментная адресация реализуется следующим образом.

ОС предоставляет программе необходимое пространство памяти в виде нескольких сегментов. Физические адреса сегментов заносятся в специальную таблицу — таблицу дескрипторов. Каждая строка таблицы содержит дескриптор. По одному дескриптору на сегмент. Дескриптор хранит адрес начала сегмента (базовый адрес), его размер (лимит) и ряд признаков (рис. 6.10).

Признаки	База сегмента	Лимит

Рис. 6.10. Строка таблицы дескрипторов

Информация о расположении таблицы дескрипторов хранится в регистре таблицы дескрипторов.

На рис. 6.11 показана общая схема образования физического адреса при сегментной адресации.

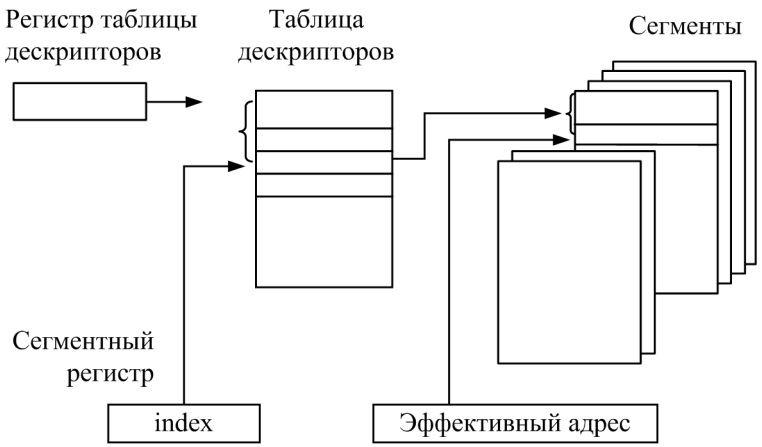


Рис. 6.11. Образование физического адреса при сегментной адресации

Полный адрес в ОП задается двумя компонентами — эффективным адресом и индексом сегмента, содержащимся в сегментном регистре. Сегментный регистр указывает на положение строки — дескриптора сегмента. База сегмента указывает на начало сегмента в памяти, а эффективный адрес — смещение относительно начала сегмента. Поле "лимит" в дескрипторе указывает на размер сегмента.

Перед вычислением физического адреса процессор проверяет, не больше ли эффективный адрес, чем лимит сегмента. Если больше — обращение к памяти не происходит, фиксируется ошибка.

Размер таблицы дескрипторов зависит от числа разрядов, используемых в сегментном регистре для поля индекса. Например, в процессорах Pentium это 14 разрядов (в 16-разрядном регистре в 2 разряда записывается дополнительная информация). Следовательно, в таблице дескрипторов должно быть 2^{14} строк по несколько байтов (в Pentium их 8). Таблиц должно быть, по крайней мере, столько, сколько одновременно программ находится в памяти. В общем, места для них надо столько, что хранить их в процессоре было бы слишком дорого, и они хранятся в основной памяти.

Это, однако, приводит к тому, что для чтения одной команды или данного из памяти придется обратиться к ней два раза — сначала, чтобы извлечь дескриптор, и лишь затем, чтобы прочитать затребованную информацию.

Избежать двойного обращения к памяти можно, если принять во внимание, что загрузка сегментных регистров происходит сравнительно редко (следствие локальности). С учетом этого в структуру процессора вводится несколько пар регистров: регистр базы сегмента + регистр лимита сегмента, по одной паре на каждый сегментный регистр.

Механизм сегментной адресации работает следующим образом.

При вызове программы в ОП операционная система создает для нее таблицу дескрипторов. Перед тем как передать управление программе, операционная система заносит адрес начала таблицы дескрипторов этой программы в базовый регистр. В программе имеются команды загрузки регистров. Исполнение такой команды состоит из двух частей. Первая — буквальное выполнение команды — запись в сегментный регистр нового значения. Вторая часть, невидимая для программиста, схематически изображена на рис. 6.12. Результат ее работы — новое содержимое регистров базы и лимита сегмента.

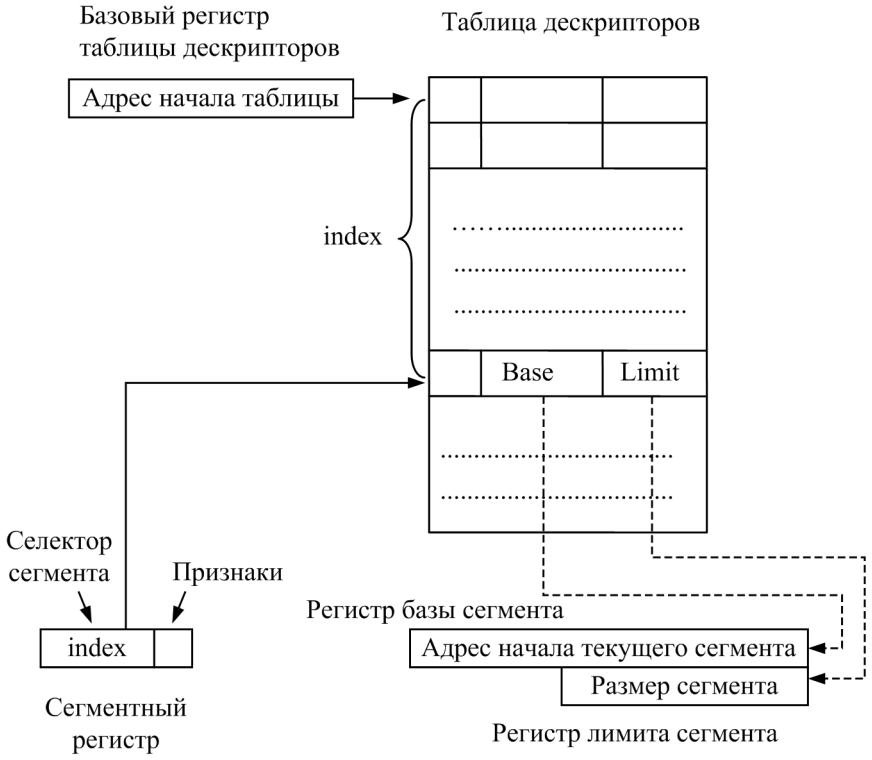


Рис. 6.12. Загрузка регистров базы и лимита сегмента

В дальнейшем значения в этих регистрах будут сохраняться, пока не произойдет новая загрузка сегментного регистра. Обращаясь к ячейке памяти, программа указывает только эффективный адрес, подразумевая, что сегмент остается тем же. Физический адрес образуется в соответствии со схемой, приведенной на рис. 6.13. При этом каждый раз осуществляется контроль, не выходит ли полученный адрес за границы сегмента.

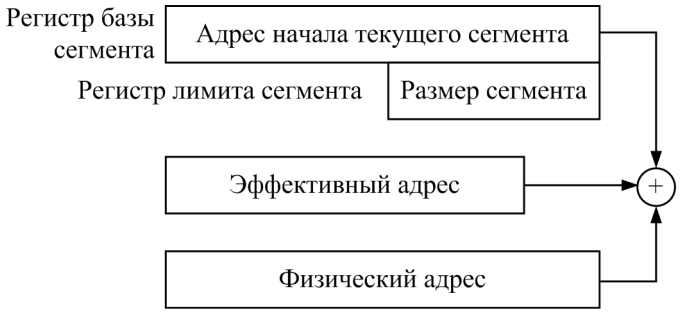


Рис. 6.13. Схема формирования физического адреса

На рис. 6.12 и 6.13 регистр лимита сегмента изображен меньшей длины, чем размер эффективного адреса. Обычно так оно и есть, тем более, что размер сегмента устанавливается не в байтах, а в страницах, т. е. содержимое регистра лимита сегмента умножается, например, на 2^{12}. Следовательно, каждый сегмент может получить адресное пространство, равное максимально адресуемой памяти.

Все сегменты одновременно могут не разместиться в памяти, а это означает, что сегмент, расположенный в памяти, может быть вытеснен другим, необходимым в настоящий момент программе. Так что наличие дескриптора сегмента в таблице еще не означает его присутствия в памяти. Поле признаков содержит специальный бит, показывающий присутствие сегмента. При записи нового значения в сегментный регистр сначала проверяется этот бит. Если сегмент в памяти — происходит загрузка регистров базы и лимита сегмента в соответствии со схемой (см. рис. 6.12). Если нет — выполнение команды прерывается, управление переходит к ОС, которая вызывает нужный сегмент в ОП, в таблицу дескрипторов записывает его адрес и лимит и изменяет значение бита присутствия. Затем выполнение команды продолжается. При вытеснении сегмента из памяти стирания ни самого сегмента, ни его дескриптора не происходит. Изменяется только его бит присутствия.

Поле признаков содержит и другую информацию, например, разрешено ли в этот сегмент производить запись.

6.2.4. Выводы по использованию виртуальной памяти

Итак, использование виртуальной памяти позволяет:

1. Эффективно использовать локальность ссылок и работать с памятью большого объема, значительно превышающего объем адресуемой памяти с быстродействием, немногим меньшим, чем быстродействие ОП.

   Доля стоимости оперативной памяти составляет порядка 10% от всей стоимости компьютера, а если бы ее объем был такой же, как у дисковой памяти, ее стоимость возросла бы в 150—200 раз!

2. Пространство виртуальных адресов программы, а при сегментной организации и ее частей, ограничено только шириной шины адреса, что обычно много больше пространства физических адресов.

3. У каждой программы свое виртуальное адресное пространство, а при сегментной адресации — и у частей (сегментов) программы.

4. Программы, одновременно размещенные в памяти, защищены друг от друга. Это обеспечивается тем, что формирование таблиц страниц и таблиц дескрипторов выполняет ОС, а формирование всех физических адресов и доступ к ним контролируется. Тем самым каждой программе доступны только ее сегменты и страницы.

Главное различие сегментной и страничной организации: сегменты — логические единицы программы и определяются при программировании, страницы — формальные единицы и механизм страничной адресации полностью скрыт от программиста.

Сегментная и страничная организация могут применяться вместе. В этом случае эффективный адрес преобразуется дважды. Сначала по содержимому сегментного регистра и эффективному адресу через таблицу дескрипторов определяется промежуточный (логический) адрес, затем — через таблицу страниц (а может, и через каталог таблиц) — окончательный, физический адрес. (Здесь нет общепринятой терминологии — любой из адресов, кроме физического, называют логическим. Что, впрочем, логично.)

6.3. Кэш-память

Кэш-память предназначена для временного хранения команд и данных. При эффективном ее использовании в ней находятся те команды и данные, которые понадобятся процессору в ближайшее время. Емкость кэш-памяти значительно меньше оперативной, в нее можно скопировать лишь часть программы, поэтому в кэше, кроме самих команд или данных, хранятся, в той или иной форме и их адреса.

Использование кэш-памяти, в общих чертах, происходит следующим образом. Когда процессору нужно прочитать команду или данное из памяти, он помещает адрес запроса в порт — регистр MAR. MMU — контроллер памяти преобразует виртуальный адрес в физический. Затем контроллер кэш-памяти пытается найти нужное данное или команду в кэше. Если находит (эту ситуацию называют попаданием), то производит чтение из кэша, а если нет (промах), — обращается к основной памяти, извлекает данное или команду и помещает его в кэш, после чего, или одновременно, пересылает данное или команду процессору. Так как промахи происходят редко, применение кэш-памяти позволяет доставлять процессору данные и команды значительно быстрее.

Запись данных в память также может производиться через кэш, хотя есть варианты организации кэша с записью непосредственно в память.

Существуют два принципа организации кэш-памяти: прямое отображение и ассоциативное. Ассоциативное — более эффективное, но и более дорогое. Поэтому имеются три разновидности кэш-памяти: кэш прямого отображения, ассоциативный кэш (небольшой, ввиду дороговизны) и множественно-ассоциативный (наборно-ассоциативный), основанный на комбинации двух принципов.

6.3.1. Кэш прямого отображения

Когда изучают размещение в памяти программ и данных, пользуются линейным схематическим изображением памяти. Обычно рисуют вертикальную последовательность байтов, растущую снизу вверх или сверху вниз. Здесь принят второй вариант (рис. 6.14).

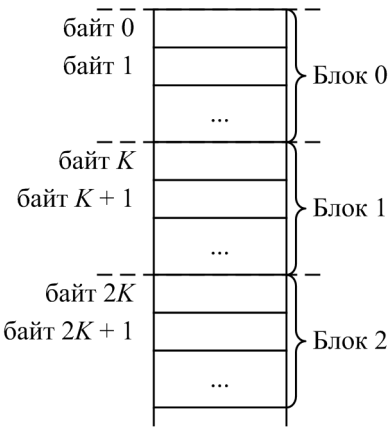


Рис. 6.14. Линейная схема памяти

Для изучения взаимодействия кэш-памяти и основной памяти удобно использовать двухмерную схему кэш-памяти и трехмерную — основной. Кэш-память прямого отображения делится на строки одинаковой длины — последовательность некоторого числа байтов (16—128, возможно и больше). Байты в строках изображаются в горизонтальной последовательности, строки — в вертикальной (рис. 6.15). Будем считать, что строка имеет длину $K = 2^k$ байт, размер кэша $N = 2^n$ байт. Тогда кэш содержит $L = 2^{n-k}$ строк.

Блок 0	байт 0	байт 1	...	байт $K-1$
Блок 1	байт K	байт $K+1$	...	байт $2K-1$
Блок 2	байт $2K$	байт $2K+1$	...	байт $3K-1$
	...			

Рис. 6.15. Двумерная схема кэш-памяти

Основная память разбивается (условно) на такие же строки. Их принято называть блоками. На рис. 6.16 представлена двумерная схема основной памяти.

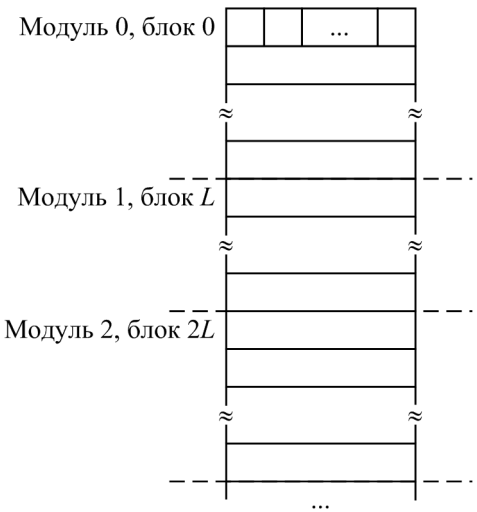


Рис. 6.16. Двумерная схема основной памяти

Плоскую память условно делим на модули. Размер модуля равен размеру кэш-памяти. Трехмерная схема памяти получается, если расположить модули параллельно, как на рис. 6.17.

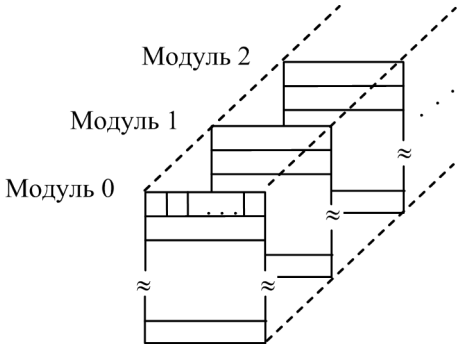


Рис. 6.17. Трехмерная схема памяти

В такой схеме адрес памяти, состоящей из $M = 2^m$ ячеек, можно представлять состоящим из трех частей (координат трехмерного изображения) как на рис. 6.18.

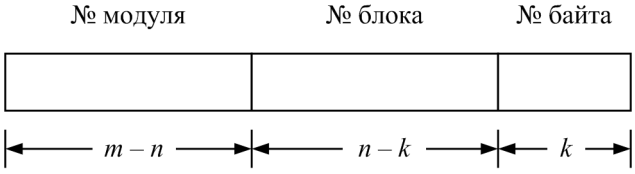


Рис. 6.18. Координаты трехмерного изображения

Обмен данными между основной и кэш-памятью производится целыми блоками.

Например. Кэш-память — 128 Кбайт = 2^{17} байт. Размер строки — 32 байта. Тогда вся кэш-память будет состоять из 2^{12} строк. Оперативная память объемом 512 Мбайт = 2^{29} байт состоит из 2^{12} модулей.

Горизонтальными плоскостями набор модулей разбивается на слои. В каждый слой попадают блоки с одинаковыми номерами внутри модулей. В слой 0 попадают все блоки № 0, в слой 1 — все блоки № 1 и т. д. На рис. 6.19 заштрихованы блоки одного из слоев. Блоки, входящие в слой i, сопоставляются строке i кэша. Любой из блоков оперативной памяти, входящих в слой i, может быть скопирован в строку i кэша.

Кэш-память состоит из двух частей — банка тегов (каталога) и банка кэша. Каждой строке банка сопоставлена одна строка каталога.

Строка i каталога хранит информацию о том, копия какого именно блока слоя i находится в настоящий момент в банке кэша. Каждый тег состоит из

двух полей. Одно поле предназначено для хранения номера модуля, блок которого отображен в настоящий момент в строке банка, второе поле — для хранения признаков (рис. 6.20).

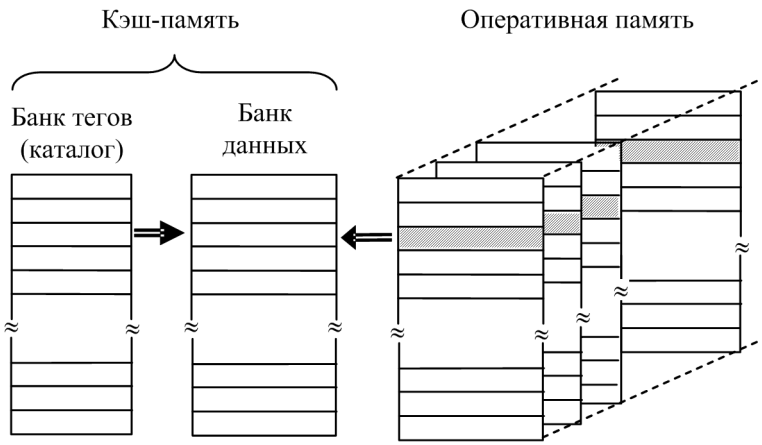


Рис. 6.19. Слои оперативной памяти

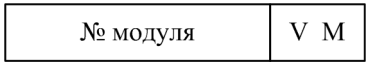


Рис. 6.20. Структура тега

Признаки здесь те же, что описаны выше: V — признак действительности (присутствия), M — признак модифицированности.

$V = 0$ означает, что содержимое строки кэша не соответствует никакому блоку памяти. Например, при смене программы кэш должен быть освобожден. Строки физически не стираются, а объявляются недействительными — в каталоге для всех строк устанавливается $V = 0$. По мере заполнения кэша в нем появляются строки с $V = 1$.

$VM = 10$, если строка кэша является копией блока памяти, номер которого хранится в теге, $VM = 11$ — в кэше хранится строка, скопированная из памяти, а затем модифицированная, т. е. строка кэша не совпадает с соответствующим блоком памяти, причем в кэше — новое, правильное значение.

Когда контроллер обращается к кэшу, он делит адрес памяти на три части: a — номер модуля, b — номер строки (индекс в каталоге) и disp — номер байта в строке. По индексу b он находит нужный тег и сравнивает адресную часть тега (α) со старшей частью адреса (a). При совпадении, если к тому же

$V = 1$, находит нужный байт в банке кэша, отсчитывая смещение disp от начала строки (рис. 6.21). Процессор может обратиться к байту памяти, слову или другой единице информации. Это никак не меняет принципов его взаимодействия с памятью. Для краткости будем в дальнейшем говорить о словах.

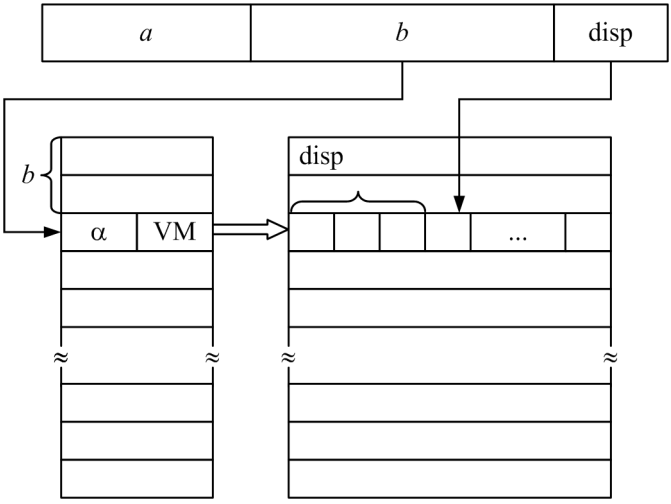


Рис. 6.21. Использование частей адреса при обращении к кэшу

Чтение из кэша

При чтении контроллер кэша по средней части адреса соединяется с соответствующей строкой каталога. Дальше происходит либо "попадание" — в кэше имеется необходимая информация, либо — "промах" и информацию надо извлекать из основной памяти. Точнее, может возникнуть один из следующих четырех случаев.

1. Строка банка кэша содержит требуемую информацию ($a = \alpha$ и $V = 1$).
2. Строка банка кэша не содержит достоверной информации (признак достоверности $V = 0$).
3. Строка банка кэша содержит немодифицированную информацию из другого модуля памяти ($a \neq \alpha$, $V = 1$ и признак модификации $M = 0$).
4. Строка банка кэша содержит модифицированную информацию из другого модуля памяти ($a \neq \alpha$, $V = 1$, в строку производилась запись — $M = 1$).

Случай 1 соответствует попаданию, и надо просто прочитать нужное слово из строки банка кэша. Случаи 2, 3 и 4 соответствуют промаху. При промахе на-

до обратиться к основной памяти и заменить строку в кэше. После чего возникнет ситуация случая 1. Но в четвертом случае надо предварительно выгрузить модифицированную строку в основную память.

Предположим, пришел запрос на чтение.

На рис. 6.22 — состояние входного регистра и строки b каталога кэша.

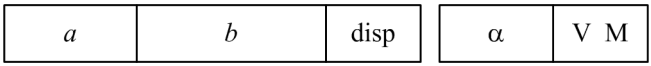


Рис. 6.22. Входной регистр и строка каталога

Схематически работу контроллера кэша при чтении можно изобразить так, как показано на рис. 6.23.

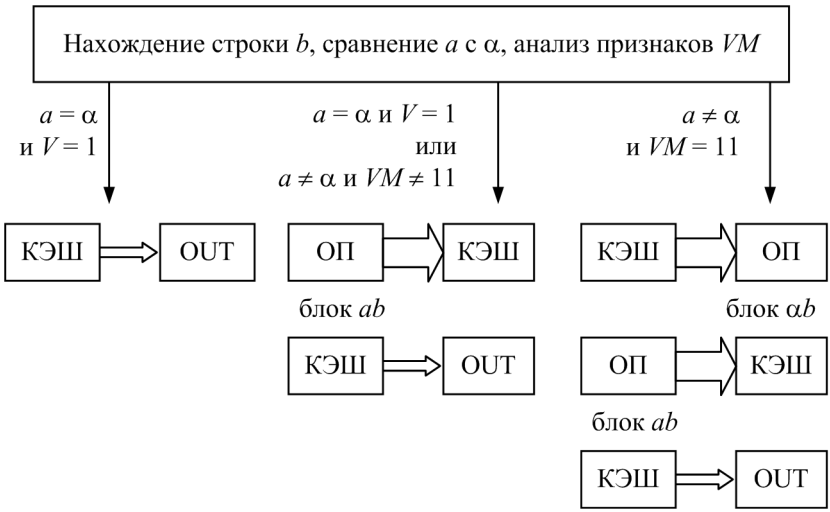


Рис. 6.23. Работа при чтении

Левая стрелка — попадание. Слово из кэша передается в выходной регистр (OUT). Обычно применяется "агрессивное" чтение (используется также термин "спекулятивное" чтение), т. е. чтение начинается сразу, как только сформируется адрес слова, без ожидания результатов сравнения в каталоге. Если произойдет промах, чтение прерывается, а частичные результаты аннулируются. Если попадание — результат получается быстрее. Недостатки агрессивного чтения — более сложная организация контроллера и напрасное энергопотребление при промахе. А это — дополнительный нагрев процессора, требующий дополнительного охлаждения.

Средняя стрелка — промах. В строке b банка кэша находится ненужная информация (если $V = 0$) или копия блока оперативной памяти из модуля α (если $V = 1$, $M = 0$). В любом случае информацию в строке банка кэша можно стереть. В этом случае блок b из модуля a оперативной памяти копируется в строку b банка кэша (широкая стрелка призвана подчеркнуть, что копируется именно блок — много слов). В банке тегов (в его строке b) значение α заменяется на a, записываются значения признаков $VM = 10$. Затем слово (узкая стрелка) передается из кэша в выходной регистр.

Правая стрелка — тоже промах. Но теперь строка b банка кэша хранит полезную информацию — модифицированный блок модуля α. Поэтому сначала строка кэша выгружается в память, а затем выполняются действия, описанные выше.

Возможен более быстрый вариант при промахе: слово в OUT засылается одновременно с засылкой блока в кэш (рис. 6.24). Но он более сложно реализуется.

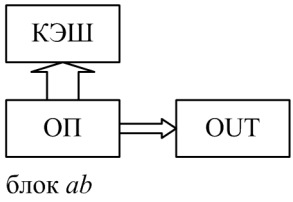


Рис. 6.24. Одновременное копирование блока и чтение слова

Запись в кэш

При записи используется ряд различных вариантов.

Во-первых — обратная запись. Этот вариант предусматривает, что запись всегда производится в кэш. Если нужной строки в кэше нет, то она пересылается из основной памяти, после чего производится запись в кэш. Если в кэше в этот момент находится модифицированная запись, то она предварительно копируется в основную память.

Схема действий при обратной записи подобна схеме действий при чтении (рис. 6.25). При этом всегда устанавливается $VM = 11$. Здесь IN — входной регистр кэша.

Во-вторых — сквозная запись. В этом случае в обязательном порядке производится запись в память, поэтому признаки V и M не используются.

Реализуются различные варианты сквозной записи.

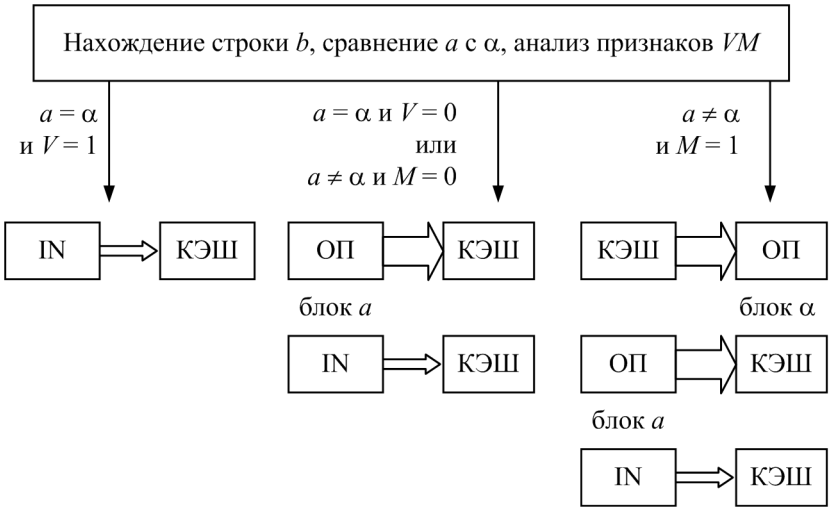


Рис. 6.25. Схема действий при обратной записи

При попадании запись возможна по двум схемам. Первая — с одновременной записью в кэш и в память (рис. 6.26, слева). Вторая — буферизированная сквозная запись. Сначала производится запись в кэш и в буфер, а после из буфера — в память (рис. 6.26, справа). Буфер работает со скоростью кэша, поэтому запись производится быстро, процессор продолжает работать, не дожидаясь записи в память.

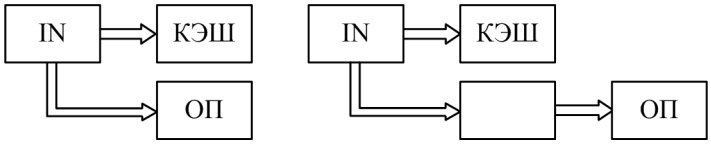


Рис. 6.26. Сквозная запись при попадании.
Справа — буферизированная сквозная запись

При промахе тоже есть два варианта, показанных на рис. 6.27. Первый (слева) — "сквозная запись без отображения". Запись только в память. При таком устройстве в кэш попадают данные только при чтении. Справа — второй вариант — "сквозная запись с отображением". Запись слова в память, а затем — копирование строки в кэш.

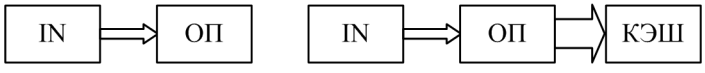


Рис. 6.27. Сквозная запись при промахе. Справа — сквозная запись с отображением

Получается, что существование кэша, в случае сквозной записи, дает некоторый выигрыш лишь при попадании и использовании схемы буферизированной сквозной записи.

Однако на производительности компьютера это отражается мало. Дело в том, что доля обращений по записи мала по сравнению с чтением. Во-первых, команды только читаются. Во-вторых, одни и те же данные обычно читаются многократно. По разным оценкам обращения по записи составляют от 7 до 30% от общего числа обращений по данным.

Кроме того, в многопрограммном режиме возможно обращение к памяти помимо процессора (прямой доступ к памяти). Тогда согласованность данных в кэше и в основной памяти приобретает важное значение. А режим сквозной записи такую согласованность гарантирует.

Так что более медленные, но простые схемы вполне конкурентоспособны.

Для уменьшения числа промахов применяется упреждающая выборка. Она заключается в том, что после прочтения в кэш блока из основной памяти контроллер проверяет, действительна ли следующая строка. Если строка свободна ($V = 0$), в нее записывается следующий блок из памяти (запись производится в "свободное время", одновременно с работой процессора). Это особенно эффективно в начале работы новой программы.

Секторированный кэш

Существует особая разновидность кэша прямого отображения — секторированный кэш. В нем строка кэш-памяти увеличена и может хранить копии нескольких подряд расположенных блоков основной памяти, например, четырех. Строка делится на части — секторы, так чтобы в сектор помещался один блок. Интерпретация адреса памяти для секторированного кэша изображена на рис. 6.28.

При обращении к памяти: a, по-прежнему, — номер модуля памяти, bc — номер блока, disp — смещение в блоке. При обращении к кэшу: b — номер строки, c — номер сектора, disp — смещение в секторе. Для попадания a должно совпасть с α.

Смысл разделения на секторы: строка может не загружаться целиком, а отдельными блоками, каждый из которых имеет в теге свой признак присутствия. Это позволяет работать, не дожидаясь загрузки всей строки. Остальные секторы могут быть загружены по мере обращений или в порядке упреждающей выборки.

Серьезным недостатком кэш-памяти прямого отображения является невозможность одновременного хранения копий слов, принадлежащих одному слою. Например, при кэш-памяти 1 Мбайт основная память в 1 Гбайт состоит из 1024 модулей. То есть на каждую строку кэш-памяти имеется по 1024 бло-

ка-претендента. Если программа поочередно обращается к словам из разных модулей, отражаемым в одно и то же слово кэш-памяти, то будут происходить постоянные промахи и постоянная перезагрузка кэш-памяти.

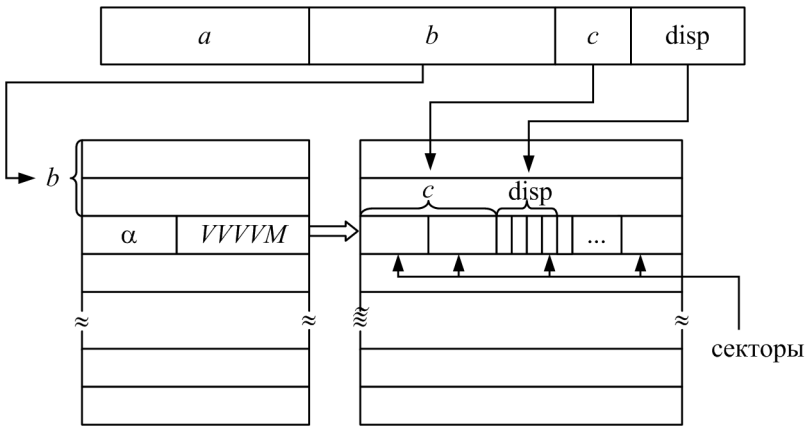


Рис. 6.28. Адрес памяти для секторированного кэша

6.3.2. Ассоциативный кэш

Ассоциативный кэш способен одновременно хранить любые комбинации блоков. Как и кэш прямого отображения, он состоит из двух частей — каталога и банка кэша. Однако его каталог хранит полный адрес блока. Запрашиваемый адрес памяти делится на две части — адрес блока и смещение слова в блоке (рис. 6.29).

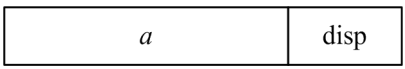


Рис. 6.29. Интерпретация адреса в ассоциативном кэше

Каталог имеет N входов (N — число строк в кэше). Адрес строки подается одновременно на все входы и сравнивается с каждой строкой каталога. Если в какой-то строке произошло совпадение (попадание), то по значению disp в строке отсчитывается нужное смещение. Если выполняется чтение, информация передается из банка в выходной регистр, при записи — в обратном направлении (рис. 6.30).

При промахе в кэш вызывается тот блок, к которому произошло обращение (его адрес в основной памяти $a00\ldots0$). Сначала контроллер ищет свободную строку в банке кэша, в которую будет записан новый блок памяти. Если есть

строка с $V = 0$, то копирование производится в нее. Если такой строки нет, то замещается одна из действительных строк. Выбор замещаемой строки, как и в таблице страниц виртуальной памяти, производится по результатам анализа значений R в строках каталога. Чаще всего замещается наиболее давно использовавшаяся строка.

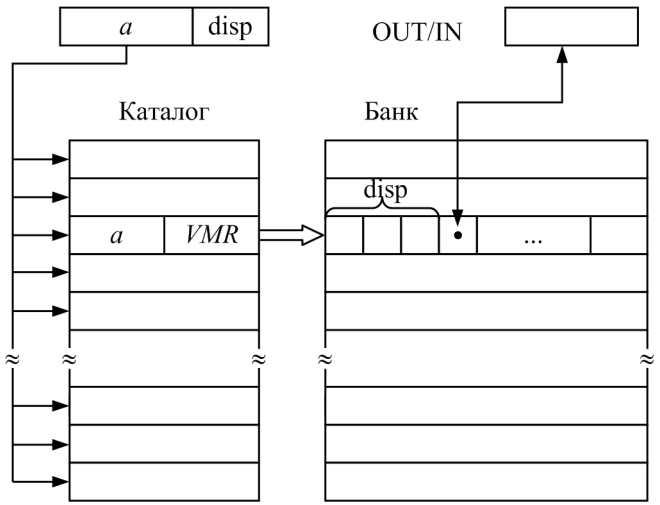


Рис. 6.30. Работа ассоциативного кэша

Варианты политики записи и чтения в ассоциативном кэше такие же, как и в прямом.

6.3.3. Множественно-ассоциативный кэш

Как уже говорилось, множественно-ассоциативный кэш — компромисс между прямым и ассоциативным кэшами. Так же как в ассоциативном кэше здесь имеется много входов, так же как в прямом — блок оперативной памяти может копироваться не в любую строку кэша. На рис. 6.31 — схема двухвходового множественно-ассоциативного кэша.

N-входовый кэш имеет N банков и столько же каталогов.

Как и в ассоциативном кэше здесь блок основной памяти может быть записан не в одну строку кэша, но эта возможность ограничена числом банков. То есть одному слою сопоставляется N блоков — по одному в каждом банке и блок основной памяти может быть записан в любой из банков.

Таким образом, для совокупности блоков b модулей основной памяти совокупность строк b кэша является ассоциативной памятью верхнего уровня. А все вместе — набор ассоциативных кэшей.

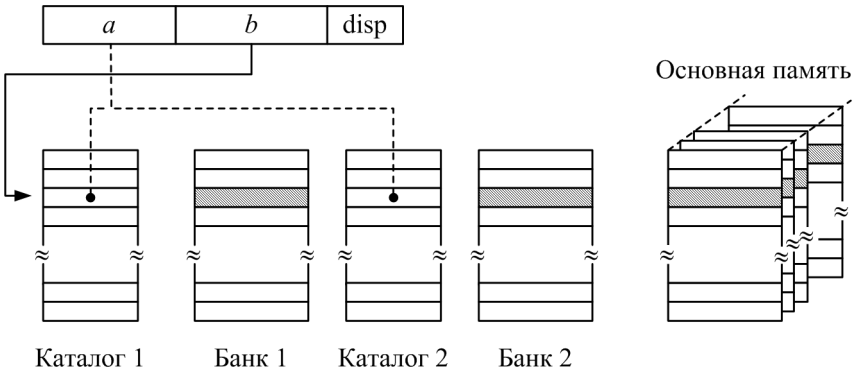


Рис. 6.31. Множественно-ассоциативный кэш

Для наборно-ассоциативных кэшей действительно все, что сказано о прямом и ассоциативном кэшах.

При запросе по индексу (средняя часть адреса) определяется слой и его ассоциативный кэш. Поиск нужной строки с заданным индексом производится одновременно по всем каталогам. В случае промаха замена производится с учетом значения признака R. Политики записи и чтения — те же, что и в прямом кэше.

Как уже говорилось, в современных компьютерах используется многоуровневая кэш-память. Минимум два уровня — L1 и L2.

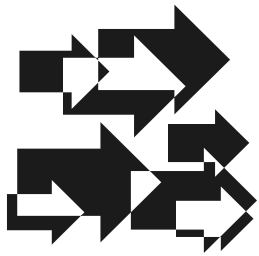
Обычно L1 расположена на одном кристалле с процессором, поэтому она небольшая — 8, 16, 32 Кбайт. Чаще всего это два раздельных кэша: кэш команд и кэш данных. Вообще-то смешанный кэш, если его емкость равна сумме емкостей раздельных кэшей, эффективней. Иногда чаще происходит чтение команд, а иногда данных. В смешанном кэше нужная пропорция регулируется автоматически, самими запросами. В раздельном кэше размер памяти команд и данных фиксирован. Однако устройство раздельных кэшей проще. В кэше команд, например, можно учесть то обстоятельство, что к нему не бывает обращений по записи. А главное, что выборку команд и данных из раздельных кэшей можно вести одновременно. В качестве кэша первого уровня обычно применяются наборно-ассоциативные кэши на 4, 8 и более входов.

Кэш второго уровня чаще всего расположен вне кристалла процессора, но очень близко к нему (иногда, все же, в самом кристалле). Его емкость обычно 128, 256, 512 Кбайт. Обычно это тоже наборно-ассоциативные кэши.

Как уже говорилось, бывают кэши L3 и L4.

Выше было описано взаимодействие кэша с основной памятью. Однако если везде заменить слова "кэш" на Li, а "основная память" на L(i+1), все изложенное, кроме, возможно, мелких деталей, окажется верным.

ГЛАВА 7



Организация процессора

7.1. Конвейер команд

7.1.1. Организация конвейера

Для организации конвейера необходимо внести два принципиальных изменения в конструкцию процессора.

1. Обработка команды разделяется на отдельные независимые шаги. Обрабатывающее устройство, прежде выполнявшее все эти шаги, разделяется на отдельные, действующие согласованно блоки, каждый из которых способен выполнить один из шагов обработки.

2. Вводятся специальные буферные запоминающие устройства в качестве посредников между блоками. Это позволяет блоку, закончившему свой шаг, передать результаты в буфер и немедленно приступить к обработке следующего задания, не дожидаясь окончания работы следующего блока.

Процесс выполнения команды арифметико-логического типа естественно разбивается на последовательность, например, таких шагов: вызов очередной команды, дешифрирование кода операции, вычисление адресов, вызов операндов, выполнение действия над операндами, засылка результата в числовой регистр или в память. Некоторые из этих шагов можно разбить на более мелкие части, а можно, наоборот, часть шагов объединить. В разных процессорах эта задача решается по-разному. Цель разбиения на шаги — построение конвейера команд. Для этого процессор делится на отдельные блоки, каждый из которых, независимо от других, выполняет один шаг команды.

Число шагов (ступеней конвейера) в разных машинах различно, иногда 10 и больше (например, в процессоре Pentium 4 их 20).

Обычно работу конвейера команд исследуют на упрощенных примерах.

Иногда на примере двухступенчатого конвейера. Первая ступень — чтение команды и операндов, вторая — выполнение операции и запись результата.

На рис. 7.1 показана структура четырехступенчатого конвейера.

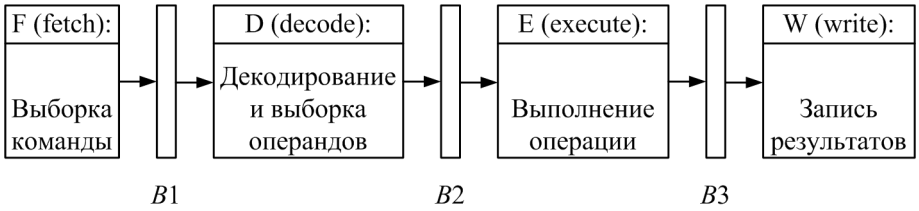


Рис. 7.1. Четырехступенчатый конвейер

Между обрабатывающими блоками F, D, E, W находятся буферы — $B1$, $B2$, $B3$. Они позволяют "развязать" отдельные блоки. Блок освобождается, как только он передал полученные результаты в буфер и может выполнять новую задачу, не дожидаясь окончания работы следующего блока.

На рис. 7.2 представлена схема работы четырехступенчатого конвейера при выполнении последовательности команд.

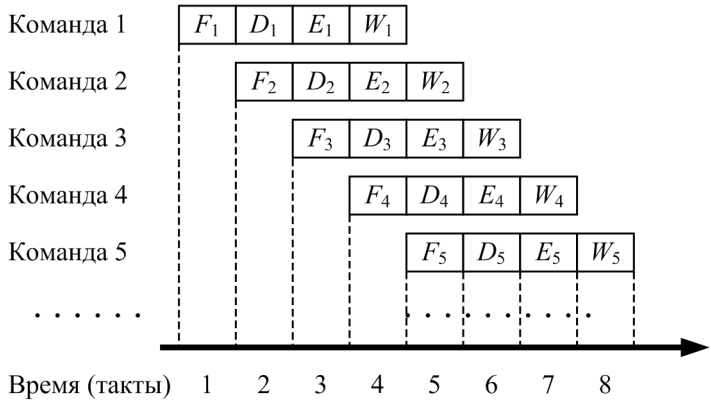


Рис. 7.2. Последовательность команд на четырехступенчатом конвейере

По оси ординат откладывается время в тактах. Такт в компьютере — это квант, неделимый отрезок времени. Любое действие в компьютере выполняется только за целое число тактов. Если оно фактически закончится, например, за полтакта, следующее действие сможет начаться все равно только по

истечении такта. Такт подбирается конструктором компьютера так, чтобы большинство элементарных действий (шагов конвейера) могло закончиться за один такт. Все же некоторые действия выполняются за два, три такта, иногда и больше.

В соответствии со схемой, на первом такте выполняется первый шаг: выборка команды 1. В конце шага команда помещается в буфер $B1$.

На втором такте работают одновременно два устройства: F и D. Устройство D забирает команду 1 из буфера $B1$, производит ее декодирование и выборку операндов. В конце такта код операции, операнды 1 и 2, а также адрес результата помещаются в буфер $B2$. Одновременно устройство F выбирает команду 2 и в конце такта помещает ее в $B1$.

На третьем такте работают три устройства. Устройство выполняет первую команду, ее результат и адрес в памяти помещаются в буфер $B3$. Одновременно устройство D переходит к обработке второй команды, а устройство F выбирает команду 3.

Начиная с такта 4, конвейер полностью загружен. Устройство W записывает в память или в регистр результат первой команды, устройство E выполняет команду 2, устройство D дешифрирует команду 3 и извлекает ее операнды, устройство F выбирает четвертую команду.

По окончании четвертого такта появляется первый результат работы конвейера — полностью выполнена одна команда. После пятого такта оказывается выполненной еще одна команда, после шестого — еще одна. То есть, начиная с четвертого такта, конвейер выполняет по одной команде за каждый такт. Таким образом, если не считать времени, затраченного на "разгон" конвейера, четырехступенчатый конвейер увеличивает быстродействие процессора в четыре раза. Очевидно, что конвейер из N ступеней способен увеличить быстродействие в N раз.

Однако такой результат в действительности недостижим. Имеются две группы проблем, значительно уменьшающих реальное быстродействие конвейера. Проблемы первой группы проявляются при выполнении последовательности команд арифметико-логического типа как задержка в работе конвейера, проблемы второй группы связаны с командами переходов и проявляются как необходимость нового разгона конвейера.

7.1.2. Задержки конвейера

Причины задержки конвейера:

- задержка работы устройств — выполнение некоторых шагов более чем за один такт;
- конфликт по ресурсам;

- явный конфликт по данным;
- скрытый конфликт по данным (побочные эффекты).

Задержка работы устройств

Выполнение любого из шагов четырехступенчатого конвейера может затянуться и тогда одна команда задерживает все остальные.

На рис. 7.3 показан случай, когда на чтение команды уходит более одного такта. Это может случиться, если нужной команды не оказалось в кэш-памяти (реально задержка произойдет не на 3, как на рисунке, а примерно на 10 тактов). Тогда конвейер останавливается и ждет освобождения устройства F.

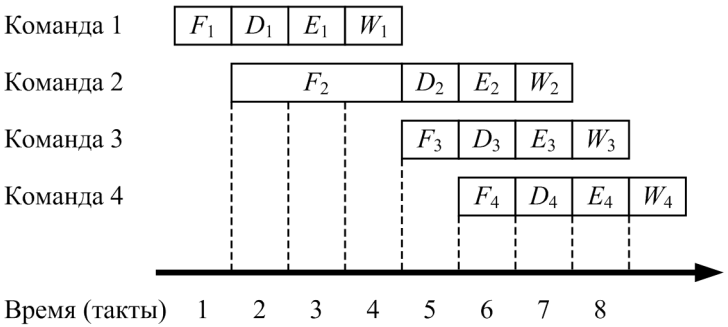


Рис. 7.3. Задержка чтения

Этап декодирования и извлечения операндов также может затянуться. Один из операндов (или оба) может отсутствовать в кэше. На рис. 7.4 шаг D второй команды выполняется в течение трех тактов — третьего, четвертого и пятого. На третьем такте одновременно выполняется этап F команды 3 и в конце такта в буфер $B1$ помещается выбранная команда. Понятно, что на четвертом такте операция D_3 начаться не может — устройство D занято. Но и шаг F_4 начаться не может — занят буфер $B1$. Команда 3 "зависает" — начавшись, не может продолжиться. Задерживается и выполнение всех последующих команд. Говорят, что в конвейере образуется "пузырь": команда 2 и предшествующая продолжают выполняться, а следом все стоит.

Выполнение арифметических операций в устройстве E тоже происходит за разное время. Если сложение и вычитание длятся примерно один такт, то умножение, а тем более деление требуют нескольких тактов (рис. 7.5).

Здесь "пузырь" еще заметнее. На схеме он продвигается сверху вниз.

Сначала конвейер работает как обычно, "в штатном режиме". Вплоть до четвертого такта. На этом такте первая команда записывает в память результат (устройство W), вторая обрабатывается устройством E (выполнение операции), третья команда — устройством D (декодирование и вызов операндов), четвертая — читается в буфер $B1$ (устройство F). На пятом такте возникает "нештатная" ситуация — устройство E не успевает закончить вычисления и продолжает работать с командой 2. Выполнение команды 3 приостанавливается — ей нужно устройство E, а оно занято. Выполнение команды 4 также приостанавливается — устройство D свободно, но ему для вывода результата работы нужен буфер $B2$, который освободится только с началом выполнения E_3.

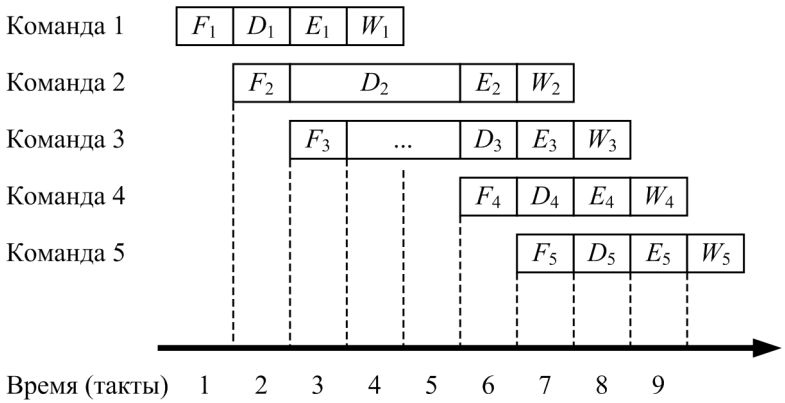


Рис. 7.4. Задержка конвейера на втором шаге

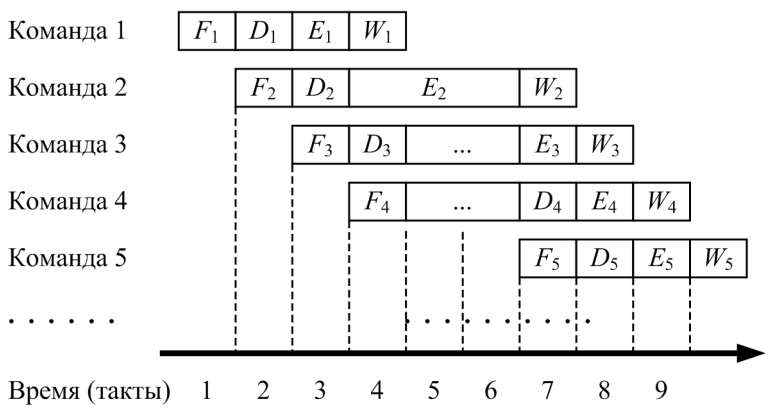


Рис. 7.5. Задержка на третьем шаге

Устройство W также может работать один такт — при записи в регистр, а может больше, — при записи в память.

На время работы устройств F, D и W большое влияние оказывает эффективное использование кэш-памяти.

Очень полезное усовершенствование конвейера — увеличение размеров буфера $B1$ и организация в нем очереди команд. Устройство F, закончив вызов команды, может не считаться с тем, что следующее за ним устройство декодирования еще не закончило работу и не освободило буфер. Устройство F вызывает следующую команду и записывает ее в конец очереди (такая работа устройства F называется *упреждающей выборкой*). Устройство D, когда закончит декодирование команды, прочитает следующую из начала очереди. Если устройство E будет выполнять команду более, чем за один такт, очередь в буфере будет расти. Но если устройство F задержится с вызовом команды (ее не оказалось в кэше), то это не вызовет задержки конвейера. Просто уменьшится очередь.

Другие методы гораздо сложнее.

1. Разработка процессоров с несколькими конвейерами команд.
2. Разработка процессоров с такой системой команд, в которой все операции выполняются за равное число тактов (RISC-процессоры) или процессоров с RISC-ядром.

Эти, более сложные, методы будут рассмотрены позже.

Конфликты по ресурсам

Для рассмотрения конфликтов по ресурсам используем более подробную схему конвейера (рис. 7.6).

Двойные стрелки показывают порядок работы обрабатывающих блоков. Пунктирные — направления передачи информации.

Конфликт по ресурсам возникает, когда два устройства одновременно обращаются к одному ресурсу. Одному из них приходится "встать в очередь".

По существу, описанные выше задержки конвейера из-за того, что некоторые действия устройств требуют больше одного такта, тоже можно считать конфликтами по ресурсам. Это конфликты за устройства. Но обычно конфликтами по ресурсам считают одновременное обращение двух (а то и трех) устройств к памяти или к регистрам.

Как видно из схемы, к памяти четырехступенчатого конвейера одновременно могут обратиться три устройства: F, D и W. К регистрам — два: D и W.

Конфликт при одновременном обращении к регистрам возникает из-за того, что доступ ко всем регистрам происходит через один общий порт.

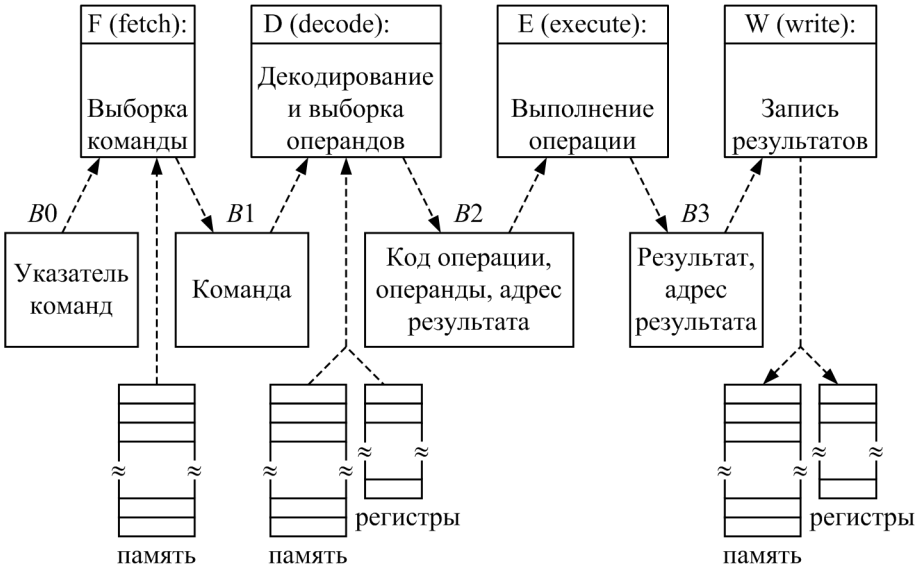


Рис. 7.6. Схема конвейера с указанием ресурсов

В более сложных конвейерах в конфликте может участвовать и больше устройств. Например, в процессорах Pentium (первой модели) был конвейер с пятью ступенями:

- ❒ выборка команды;
- ❒ дешифрирование команды и вычисление адресов;
- ❒ извлечение операндов;
- ❒ выполнение операции;
- ❒ запись результата.

Здесь на всех ступенях, кроме четвертой, может происходить обращение к памяти и регистрам.

Меры по уменьшению числа конфликтов по ресурсам:

- ❒ число конфликтов уменьшается с введением раздельных кэшей данных и команд;
- ❒ в процессорах с двумя портами регистров можно одновременно обращаться к двум регистрам;
- ❒ расслоение памяти создает возможность одновременного обращения к ячейкам из разных блоков.

Явный конфликт по данным

Данное, полученное как результат некоторой команды, если только это не конечный результат, рано или поздно будет использовано как исходное данное одной из последующих команд. Это не вызывает никаких проблем при последовательном выполнении команд. Однако если эта вторая команда расположена сразу после первой или близко от нее, одновременное их исполнение в конвейере может привести к ошибке. Такой конфликт по данным называют конфликтом типа "чтение после записи". То есть программа пишется в расчете на последовательное выполнение команд, при котором сначала должна производиться запись, а потом — чтение. Однако на конвейере чтение может произойти раньше записи, например, как на рис. 7.7.

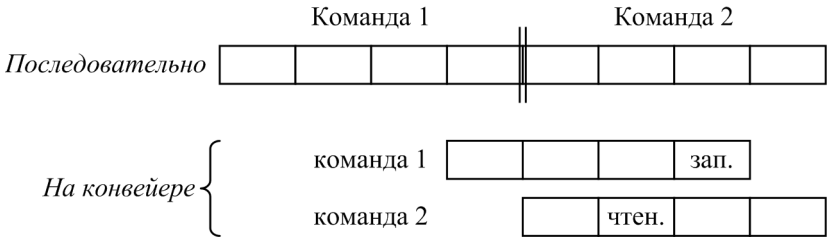


Рис. 7.7. Чтение после записи

Конфликты типа "чтение после записи" можно распознавать на этапе трансляции программы и предотвращать их.

Существуют два программных приема предотвращения конфликтов по данным.

Первый — изменение порядка следования команд на такой, при котором команды, имеющие зависимость по данным, не стоят рядом. Транслятор, как правило, легко распознает такую ситуацию. Однако это не всегда возможно. В таких случаях транслятор применяет второй прием — вставляет между зависимыми командами одну или больше команд NOP (нет операции), — команды, не делающие ничего (рис. 7.8).

После двух команд NOP команда 2 (шаг D_4) будет выбирать данные, уже записанные на шаге W_1. Этот прием приводит к задержке конвейера на два такта. Кроме того, на две команды удлиняется программа (затраты памяти).

Многие процессоры способны распознавать конфликт по данным и блокировать работу следующей команды конвейера.

Предположим, команда 1 вычисляет значение переменной, которая должна быть использована командой 2. Тогда в четырехступенчатом конвейере сложится следующая ситуация (рис. 7.9).

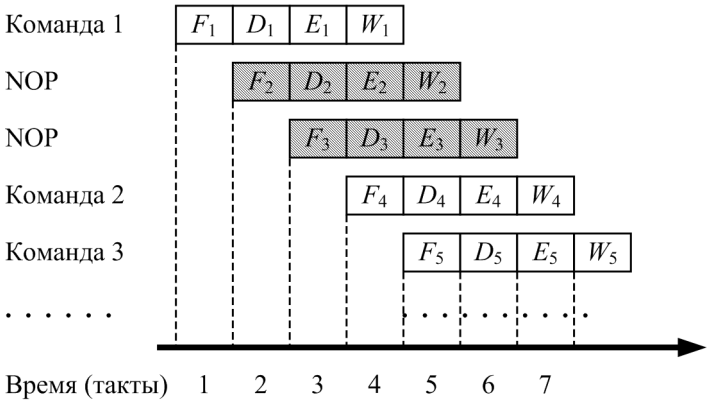


Рис. 7.8. Использование команд NOP

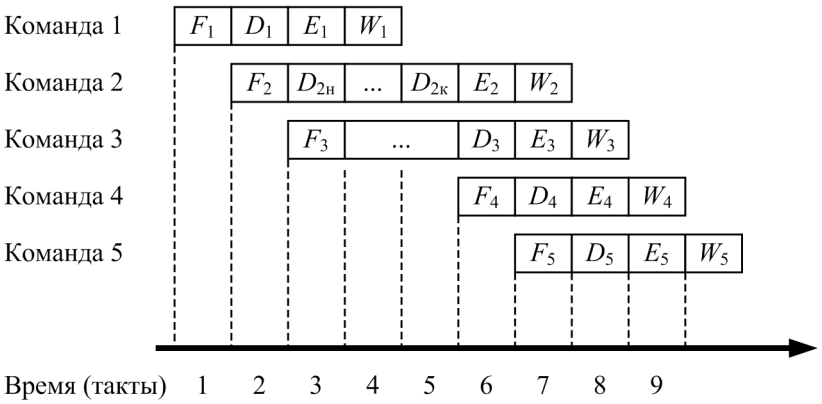


Рис. 7.9. "Пузырь"

Устройство D на третьем такте начинает обрабатывать вторую команду — дешифрировать ее, формировать адреса и извлекать операнды ($D_{2н}$), однако закончить свою работу не может, пока один из операндов не будет записан в память (или в один из регистров) — пока не закончится операция W_1. Поэтому на такте 4 устройство D простаивает и лишь на такте 5 заканчивает свою работу ($D_{2к}$). Это вызывает появление "пузыря" в команде 3 и задержку остальных команд на два такта.

Существует аппаратный метод борьбы с такими конфликтами. Он называется *продвижением операндов*. В конструкцию конвейера вносится небольшое изменение. Оно видно на рис. 7.10: от устройства E результат операции пе-

редается не только в буфер $B3$, для использования устройством W, но и в буфер $B2$, где он на следующем такте будет использован устройством E при выполнении следующей команды.

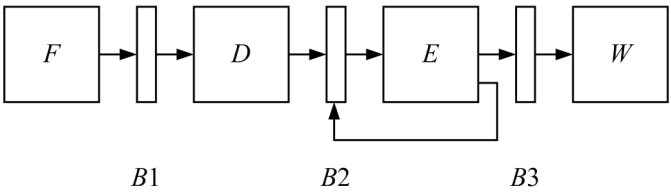


Рис. 7.10. Продвижение операндов

Скрытые конфликты по данным

Явные конфликты легко распознаются. Или заранее, на этапе трансляции, или, если адреса вычисляемые, на этапе исполнения. Например, в пятишаговом конвейере, изображенном на рис. 7.11, формальный признак конфликта — в начале очередного такта наличие одинаковых адресов в буферах $B2$ и $B3$.

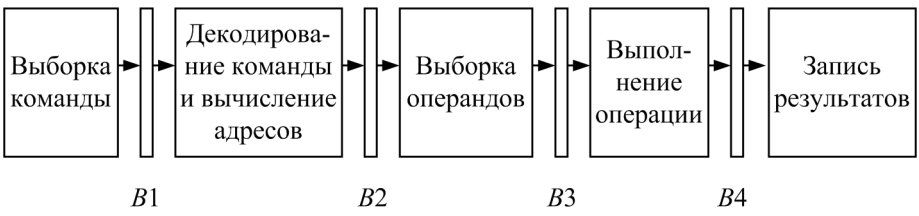


Рис. 7.11. Пятишаговый конвейер

С помощью продвижения операндов такие конфликты легко предотвращаются. Однако некоторые команды могут изменять или использовать содержимое регистров по умолчанию, без явного указания адреса.

В большинстве процессоров этим "грешат" стековые команды и все команды, вырабатывающие и использующие признак результата. Многие процессоры имеют очень удобные для программирования команды с автоинкрементом/декрементом. В таких командах операнды извлекаются (засылаются) из ячеек, адреса которых хранятся в регистрах. Команда, вместе с извлечением (засылкой) адреса из регистра, увеличивает (уменьшает) содержимое регистра на длину слова или байта, если операция байтовая. В некоторых процессорах, например, в процессорах типа Pentium, часть регистров используется по

умолчанию в качестве указателей, в качестве счетчиков числа повторений в циклах и т. д. В общем, запись в таких командах производится не только последним устройством конвейера, но и другими устройствами.

Как результат изменения содержимого регистра по умолчанию, может возникнуть скрытый конфликт по данным. Эти конфликты обычно имеют тип "запись после чтения" или "запись после записи". Они возникают в многоступенчатых конвейерах.

Для предотвращения таких конфликтов требуется настолько значительное усложнение компиляторов и аппаратуры, что основные идеи направляются в сторону изменения системы команд так, чтобы свести к минимуму число команд с побочными эффектами. В первую очередь:

- ❒ отказаться от команд с автоинкрементом/декрементом;
- ❒ отказаться от сложных способов адресации;
- ❒ ввести в команды специальный признак, разрешающий запись в регистр флагов.

7.1.3. Передача управления

Предположим, команда 1 — команда условной передачи управления. Это значит, что после ее завершения должна будет выполняться или команда 2, или целевая команда N. Схема работы конвейера в этом случае изображена на рис. 7.12.

В конвейер загружаются и начинают выполняться команды 2, 3, 4. После такта 4 выясняется, надо ли продолжать последовательное выполнение команд — 5, 6 и т. д., или перейти к выполнению команд N, $N+1$ и т. д. Если переход не произойдет, — конвейер не потеряет ни одного такта. Если произойдет — будут потеряны 3 такта. Уже начавшие выполняться команды 2, 3 и 4 из конвейера удаляются.

В более простых вариантах процессора используется технология "отложенного перехода". Конвейер не удаляет команды 2, 3 и 4 (эти команды называют *слотами задержки перехода*), и за правильность работы программы отвечает компилятор. Как и в случае конфликтов по данным, компилятор пытается вставить на их место те команды, которые надо выполнять в любом случае, независимо от возможного перехода, а если таких не найдется — пустые команды — NOP. Считается, что один слот задержки удается заполнить полезной командой в 85% случаев. Если слотов задержки больше, вероятность их заполнения невелика и технология отложенного перехода себя не оправдывает. Число слотов задержки зависит от числа ступеней конвейера. В длинных конвейерах слотов задержки больше.

Для уменьшения числа слотов задержки используется прием, аналогичный продвижению операндов: целевой адрес передается непосредственно в указатель команд (рис. 7.13).

Тогда, в случае перехода, теряются только два такта.

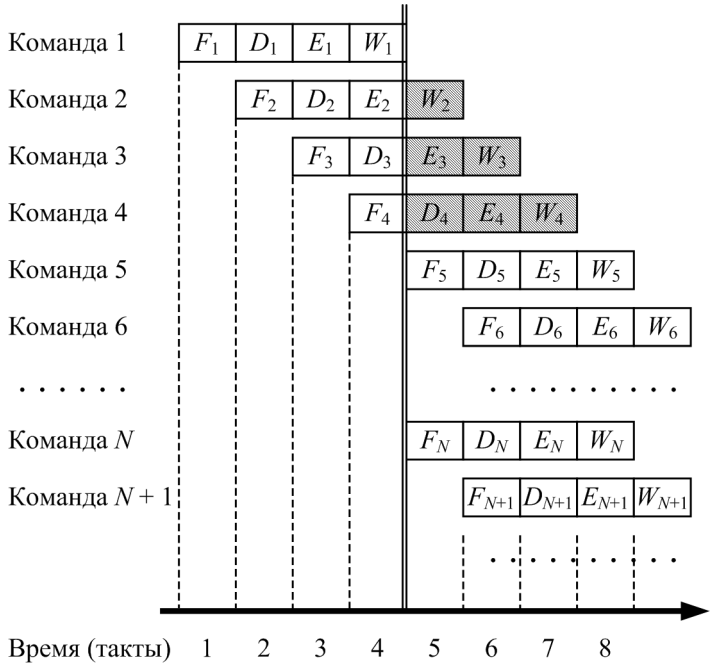


Рис. 7.12. Условная передача управления

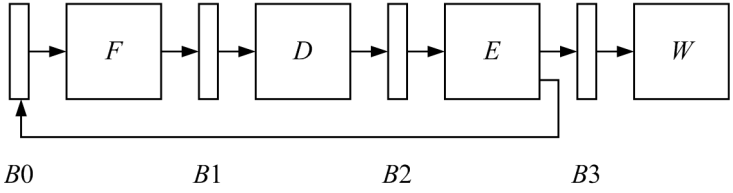


Рис. 7.13. Передача адреса перехода в указатель команд

Безусловный переход

Устройство E определяет, нужен ли переход, но сам адрес перехода вычисляется раньше, устройством D. Поэтому, в случае безусловного перехода,

целевой адрес передается уже из устройства D, вследствие чего теряется только один такт.

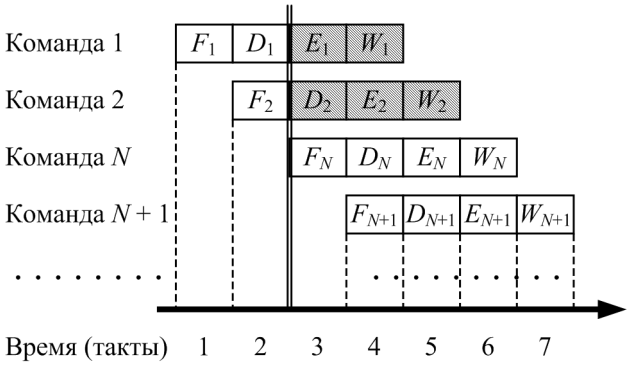


Рис. 7.14. Конвейер при безусловном переходе

Еще более дорогой вариант процессора имеет в своем составе буфер адресов перехода (Branch Target Buffer, BTB). Это небольшая кэш-память, хранящая таблицу последних переходов вида, изображенного на рис. 7.15.

Адрес команды	Адрес перехода

Рис. 7.15. Строка таблицы переходов

Устройство F, прежде чем выбрать команду, ищет ее адрес в левой части таблицы и, если находит, выбирает команду по адресу из правой части таблицы. Это позволяет для четырехступенчатого конвейера вообще не терять ни одного такта.

В некоторых процессорах имеется более сложный вариант кэша — BTIC (Branch Target Instruction Cache). В нем хранятся не только адреса точек перехода, но и команды из этих адресов.

Условный переход

При обработке на конвейере команд условного перехода, как уже говорилось, адрес возможного перехода вычисляется довольно рано, но нужен ли переход, — близко к концу конвейера. Поэтому задержка в случае перехода будет значительной. А так как команды передачи управления составляют 15—20% от общего числа команд, проблема эта весьма актуальна.

Существуют две группы методов борьбы с такими задержками.

Первая группа — параллельное исполнение обеих ветвей алгоритма. Для этого приходится дублировать часть устройств конвейера (рис. 7.16).

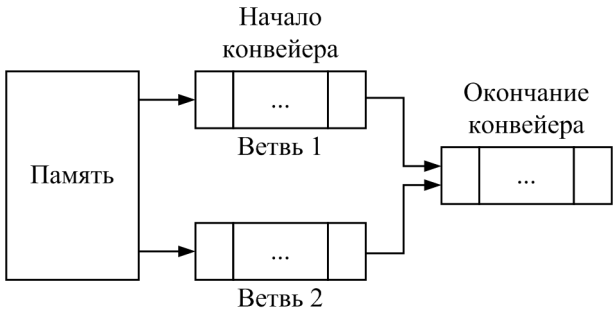


Рис. 7.16. Конвейер с параллельными ветвями

При исполнении обычных команд работает только одна ветвь конвейера. Как только устройство декодирования этой ветви обнаруживает команду условного перехода и вычисляет адрес возможного перехода, начинает работать вторая ветвь. В нее выбираются, а затем и начинают обрабатываться команды по адресу перехода. Одновременно в первой ветви продолжается выборка и начальная обработка команд, следующих за командой перехода. Команды, прошедшие в первой ветви начальную обработку, выталкиваются в общую часть конвейера. В некоторый момент будет определено, нужен ли переход. Если нет — ветвь 2 очищается и выборка в нее прекращается. Если да — очищается ветвь 1 и окончание конвейера и прекращается выборка в ветвь 1. На рис. 7.17 приведена схема выполнения фрагмента, содержащего команду условного перехода (команда 1), на пятиступенчатом конвейере. В этом конвейере ветви содержат идентичные устройства: F — вызов команды и D — декодирование команды и вычисление адресов. Общая часть конвейера состоит из трех устройств: O — вызов операндов, E — вычисление и W — запись результата.

На первых двух тактах, левее первой вертикальной линии, работает только первая ветвь конвейера. В конце второго такта определяется адрес перехода и начинает работать вторая ветвь. На тактах 3 и 4, между двойными вертикальными линиями, работают обе ветви, причем команда 1 с третьего, а команда 2 с четвертого такта, выполняются уже в общей части конвейера. В конце четвертого такта выясняется, надо ли передавать управление и, начиная с пятого такта, работает только одна ветвь. Если условие перехода не выполнено, очищается вторая ветвь и вызов в нее больше не производится. Если выпол-

нено, очищаются первая ветвь и общая часть конвейера и прекращается вызов команд в первую ветвь.

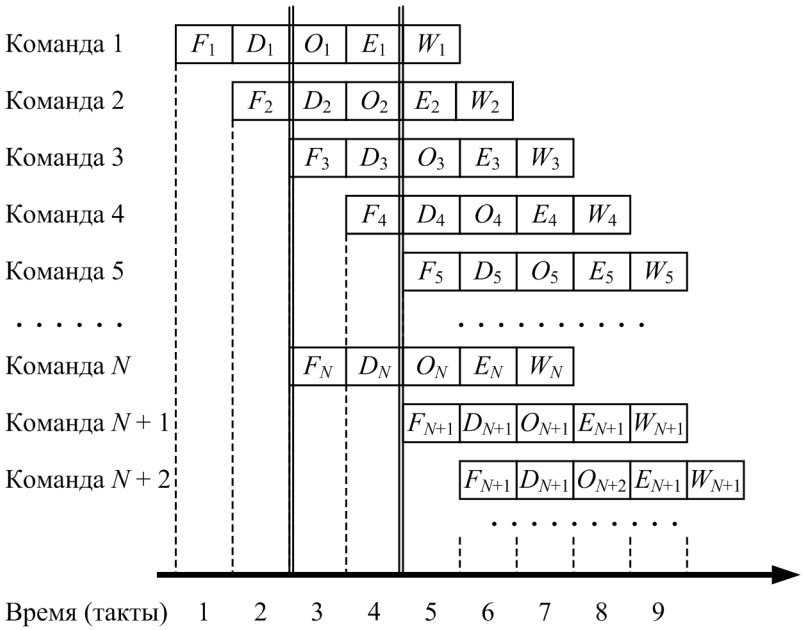


Рис. 7.17. Исполнение команд на конвейере с параллельными ветвями

Эта идея реализована в разных вариациях. Ее недостатки:

- дорого (дублируется примерно половина устройств конвейера);
- если две команды условного перехода идут подряд или очень близко, они попадают в одну из ветвей конвейера и конвейер останавливается, пока не решится вопрос с первой командой.

Вторая группа методов снижения потерь при условных переходах связана с предсказаниями переходов. Ведь если бы удалось правильно предсказывать, будет переход или нет, потери свелись бы к минимуму. Действительно, если перехода нет, то потерь нет совсем, если переход есть — адрес перехода определяется на втором такте и потери составляют один такт. А если, как описано выше, хранить в кэше таблицу переходов, то потерь не будет и при переходе.

Правда, если предсказание окажется ошибочным, а это выяснится только на одной из последних ступеней конвейера, произойдет задержка конвейера на несколько тактов (их число равно разности в тактах между моментами определения адреса перехода и факта перехода).

Методы предсказания переходов делят на статические и динамические.

Статические методы предсказывают переходы для каждой команды раз и навсегда до выполнения программы. Они основываются или на анализе кода программы на этапе компиляции, или на результатах теста. То есть предполагается, что чаще всего будет работать какая-то одна ветвь алгоритма. Например, команды управления циклом, стоящие в конце тела цикла, чаще всего передают управление "наверх". Для такой команды хорошее статическое предсказание будет: "передача управления происходит всегда". Если такой цикл всегда исполняется по 9 раз, то доля ошибок составит 10%. Используются различные принципы построения предсказаний на основе анализа кода. Например: "для всех команд переход происходит всегда" или "переход не происходит никогда". Или еще: "переход назад происходит всегда, переход вперед не происходит никогда". Есть и другие.

Некоторые методы по результатам теста определяют, что было чаще: переход или его отсутствие. Для команд, которые чаще передавали управление, устанавливается, что переход будет всегда. Для остальных — что не будет никогда.

В компьютерах, использующих статическое предсказание переходов, команды условной передачи управления имеют специальный управляющий бит, который устанавливается компилятором по результатам прогноза. Конвейер по этому биту определяет порядок выборки команд.

При использовании *динамических методов* перед каждым исполнением команды перехода делается прогноз и в соответствии с ним выбирается ветвь алгоритма.

Динамические прогнозы требуют ведения специальной таблицы. В ней для каждой команды хранится код истории переходов. Чаще всего это счетчик с насыщением. К значению счетчика при каждом переходе прибавляется единица (если в счетчике уже максимальное значение — оно не изменяется). Если перехода не произошло — единица вычитается (если там — ноль, то ноль и остается). Счетчик может состоять из одного, двух или трех бит (эксперименты показывают, что при большем счетчике вероятность верного предсказания падает). При вызове команды перехода проверяется значение ее счетчика. Если старший бит счетчика равен единице — прогнозируется переход, если нулю — прогнозируется отсутствие перехода.

7.2. Основные направления развития систем команд

7.1.1. RISC-процессоры

С самого начала компьютерной истории существовал конфликт между противоречивыми требованиями к размеру ячейки памяти. Для команд, целых

чисел, чисел с плавающей точкой, символов оптимальными были различные размеры ячейки. Постепенно область использования компьютеров смещалась в сторону обработки символьной информации. Это приводило к увеличению веса требования иметь короткую ячейку, удобную для хранения символов.

Параллельно с этим развивались сложные методы адресации, удобные для обработки массивов и более сложных структур. Различным способам адресации для записи команды требуются ячейки различных размеров.

Обе эти тенденции породили идею байтовой памяти — иметь короткую ячейку — байт, соответствующую требованиям размещения в ней одного символа, с возможностью объединения двух, четырех и более подряд расположенных байтов, для размещения в них чисел и других данных, имеющих длину более байта, а также команд различной длины.

Еще одна тенденция в развитии компьютерной деятельности. Первоначально компьютеры были чрезвычайно дороги и дефицитны. Это приводило к тому, что конструкторы компьютеров стремились максимально повысить производительность машин, не считаясь с тем, удобна ли такая конструкция для программиста (точнее, для компилятора с языка высокого уровня). Разумеется, это, в первую очередь, относится к системе команд и способам адресации. Однако постепенно доля труда программиста в конечном продукте компьютера все возрастала. Расширялась сфера применения компьютеров. Это требовало от конструкторов учета требований удобства программирования (трансляции), что также способствовало появлению разнообразных способов адресации и усложнению системы команд. Происходило "погружение приемов программирования в аппаратуру". Ярким примером этого "погружения" является аппаратный стек.

Все это происходило на фоне бурного роста объемов оперативной памяти и быстродействия процессоров, поддерживающих тенденции усложнения системы команд и способов адресации. Однако постепенно стало очевидно, что нельзя рассчитывать на бесконечный рост памяти и тактовой частоты. Настало время, когда дальнейшее повышение производительности компьютеров связано в основном с архитектурными решениями. Появились кэш-память и конвейеры команд. И тут оказалось, что сложные системы команд, наличие многочисленных форматов команд, множества способов адресации создают массу неудобств для использования конвейеров.

Сегодня существуют два направления преодоления этого противоречия. Первое — возврат к машинам с простой системой команд. При этом реализация идей параллельной обработки, упреждающей выборки и др. возлагается на "умный" компилятор. Принято говорить о RISC-компьютерах (Reduced Instruction Set Computer — компьютер с сокращенным набором команд).

Если при конструировании прежних процессоров стремились сделать их максимально удобными для трансляторов с языков высокого уровня, то RISC-процессоры конструируются максимально удобными для выполнения программ на конвейере. Выработаны принципы организации таких процессоров:

- стандартная длина всех команд, равная ширине шины данных;
- небольшое число различных команд;
- малое число способов адресации;
- малое число форматов команд;
- обращения к памяти в командах производятся не более одного раза — чтение или запись;
- большое число регистров.

Последнее требование — следствие стремления сократить число обращений к памяти. Реализация всех остальных требований приводит к значительному упрощению управляющего устройства процессора, что, в частности, позволяет разместить в нем дополнительные регистры.

В компьютерах с RISC-процессорами не менее 75% команд проходят каждое из устройств конвейера за один такт. Конечно, упрощение системы команд и способов адресации приводит к тому, что некоторые операции, прежде выполнявшиеся одной сложной командой, теперь требуют нескольких простых команд. Однако, во-первых, сложная команда вызывала задержку конвейера. И если для выполнения операции на RISC-процессоре требуется, к примеру, пять команд вместо одной, то эти простые команды выполняются в десять раз быстрее сложной. А во-вторых, сложные команды очень удобны, но употребляются они крайне редко (доли процента), а усложняют управляющее устройство примерно наполовину.

7.1.2. CISC-процессоры

Второе направление — сохранение сложной системы команд с многочисленными форматами разной длины, большим числом способов адресации при небольшом числе регистров общего назначения. Компьютеры с такой архитектурой получили название CISC (Complex Instruction Set Computer — компьютер с полным набором команд).

Цель этого направления — сохранение простоты трансляции программ с языков высокого уровня на машинный язык, сохранение программной совместимости с машинами старых моделей в сочетании с совершенствованием процессора, повышающим его производительность.

С этим направлением связаны три основных идеи: первая — суперконвейер, вторая — суперскалярный конвейер, третья — процессор с RISC-ядром.

Суперконвейер

Главное требование к любому конвейеру — добиться того, чтобы время обработки на каждой ступени было одинаковым, свести к минимуму ситуации, когда часть ступеней простаивает. Обеспечить такую равномерность легче, если уменьшить объем работ, выполняемых на каждой ступени. Конечно, за счет увеличения числа ступеней. Конвейеры, имеющие больше шести ступеней, называют *суперконвейерами*.

Главный недостаток суперконвейера — конфликты возникают даже между командами, расположенными не слишком близко друг от друга. В конвейере с 20 ступенями одновременно выполняется до 20 команд, и если число регистров общего назначения не больше 20, то наверняка всегда будет иметься конфликт по данным. Приходится использовать изощренные методы предотвращения этих конфликтов, что усложняет процессор.

Суперскалярный конвейер

Большая часть задержек конвейера вызвана разным временем работы исполнительного блока E. Особенно заметна разница между временем выполнения операций с целыми числами и с числами с плавающей запятой. На самом деле в процессоре имеются два исполнительных устройства — устройство для действий с плавающей запятой и устройство целочисленной арифметики (оно выполняет не только арифметические операции с целыми числами, но и все остальные, кроме операций с плавающей запятой). Когда работает одно из этих устройств, второе простаивает. Один из вариантов — организация на их базе единого конвейера команд с разделением потока команд на этапе исполнения на два: поток команд с плавающей запятой и поток целочисленных и других быстро исполняемых команд.

В суперскалярном процессоре имеются как минимум два параллельно работающих функциональных блока, т. е. в нем за один такт с конвейера могут сходить как минимум две команды — целочисленная и с плавающей запятой. Равномерность загрузки конвейера обеспечить легче, если иметь не два, а четыре или больше функциональных блоков (рис. 7.18).

При этом на блок диспетчеризации возлагается сложная задача: переставлять команды местами (там, где это возможно), добиваясь одновременной подачи на исполнительные устройства команд с плавающей запятой и целочисленных.

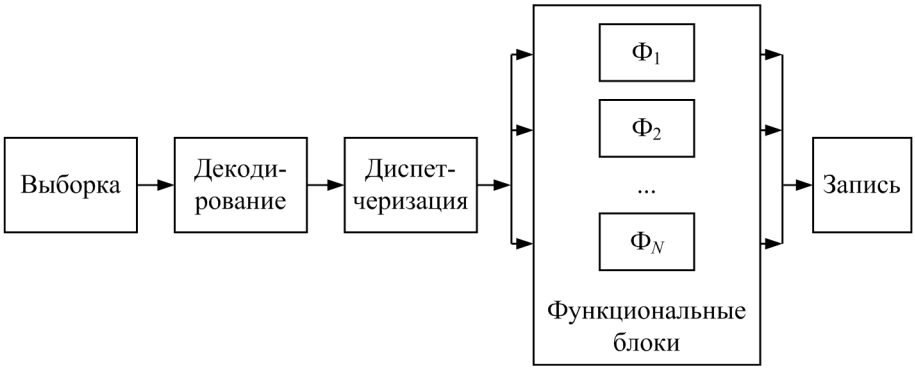


Рис. 7.18. Суперскалярный процессор

RISC-ядро

Стремление соединить достоинства CISC- и RISC-процессоров породило CISC-процессоры с RISC-ядром. В этих процессорах фактически используются два машинных языка. Один — внешний, на него опирается ассемблер, — язык, на который производится трансляция программы с языков высокого уровня. Второй — внутренний, язык RISC-ядра. Конвейер процессора разделяется на две части. Первая часть состоит из блока выборки и блока декодирования. Эти блоки принимают команды на внешнем языке. В конвейере на этапе декодирования происходит замена команд внешнего языка командами внутреннего, причем одна команда внешнего языка может заменяться несколькими командами внутреннего. На выходе блока декодирования — очередь команд на языке ядра. Функциональные блоки процессора выполняют команды именно этого, внутреннего языка.

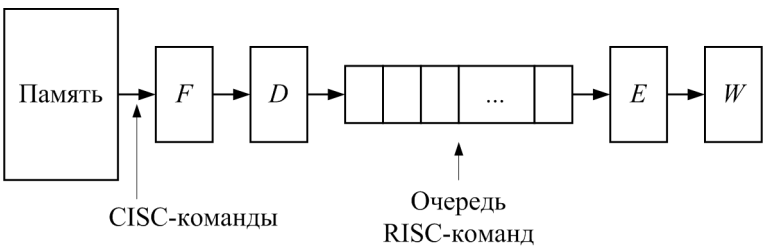


Рис. 7.19. Процессор с RISC-ядром

Обычно в одном процессоре реализуются одновременно все три идеи: процессор суперконвейерный, суперскалярный, с RISC-ядром.

7.1.3. Архитектуры с командным словом сверхбольшой длины

Другое направление, вобравшее в себя идею RISC-процессора, — VLIW-архитектура. VLIW — командное слово сверхбольшой длины (Very Long Instruction Word). Здесь выполнены все требования, предъявляемые к RISC-процессорам. Часто, даже еще более жесткие: обращение к памяти возможно только специальными командами записи/чтения. Все другие команды — регистровые. VLIV-архитектуру называют пост RISC-архитектурой. Главная новинка — *связки команд* и *пучки команд*.

Командное слово сверхбольшой длины — это несколько команд: связка (в процессоре Itanium — IA-64 их три) плюс небольшой шаблон для записи служебной информации (рис. 7.20).

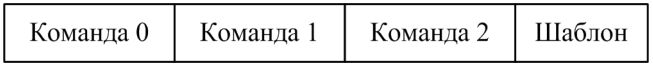


Рис. 7.20. Связка команд

Процессор рассчитан на работу с мощным компилятором, который не только транслирует программу, но и оптимизирует ее, группируя в связки команды, которые процессор способен выполнять одновременно, по числу исполнительных устройств. Например, команды сложения целых, сложения с плавающей запятой и чтение из памяти. Если все же одновременное выполнение невозможно, то в шаблоне указывается, например: сначала первую и вторую одновременно, затем — третью.

Процессор может иметь больше трех исполнительных устройств. Тогда связки команд объединяются в пучки. Команды пучка могут выполняться одновременно.

Развитие идей VLIW-архитектуры — EPIC-архитектура (Explicitly Parallel Instruction Computing — вычисления с явным параллелизмом команд).

Главное дополнение — предикация. Ее назначение — предотвращение задержек конвейера при ветвлениях программы.

Каждая команда связки включает поле предиката (рис. 7.21).

Рис. 7.21. Команда с предикатом

В состав процессора входит регистр предикатов — аналог регистра признаков. Некоторые команды могут заносить признак результата в регистр предикатов. Этот признак управляет ветвлением. При значении признака 1 должна выполняться одна ветвь программы, при значении 0 — другая. Компилятор заранее расставляет в команды значения предиката, при котором должна выполняться данная команда. Иными словами, команды одной ветви получают признак 1, другой — 0. Обе ветви исполняются одновременно на параллельных конвейерах, не дожидаясь выяснения истинности предиката. Однако запись результата блокируется до тех пор, пока в регистр предикатов не будет занесено соответствующее значение. Затем вычисления одной ветви уничтожаются, а другой — заносятся в нужные адреса.

Несмотря на очевидную прогрессивность компьютеры, опирающиеся на технологию VLIW, пока не получили широкого распространения. Основные причины этого: отсутствие качественных компиляторов и громадный объем программного и технологического обеспечения для CISC-процессоров. В качестве компромисса появились процессоры Crusoe. В них команды микропроцессора Pentium транслируются в командные слова VLIW.

Список литературы

1. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium. — СПб.: Питер, 1999. — 283 с.
2. Джермейн К. Программирование на IBM/360: Пер. с англ. — М.: Мир, 1978. — 970 с.
3. Жмакин А. П. Архитектура ЭВМ. — СПб.: БХВ-Петербург, 2008. — 315 с.
4. Кип Р. Ирвин. Язык ассемблера для процессоров INTEL: Пер. с англ. — М., СПб.: Издательский дом "Вильямс", 2005. — 905 с.
5. Кнут Д. Искусство программирования для ЭВМ: Пер. с англ. В 2-х т. Т. 2 — М.: Мир 1975. — 764 с.
6. Королев Л. Н. Архитектура электронных вычислительных машин. — М.: Научный мир, 2005. — 271 с.
7. Леонтьев В. П. Новейшая энциклопедия персонального компьютера 2003. — М.: ОЛМА-ПРЕСС, 2003. — 957 с.
8. Мураховский В. Железо ПК. Новые возможности. — СПб.: Питер, 2005. — 591 с.
9. Основы современных компьютерных технологий / Под ред. А. Д. Хомоненко. — СПб.: Корона принт, 1998. — 448 с.
10. Соломенчук В. Г., Соломенчук П. В. Железо ПК 2010. — СПб.: БХВ-Петербург, 2010. — 448 с.
11. Таненбаум Э. Архитектура компьютера: Пер. с англ., 4-е изд. — СПб.: Питер, 2006. — 698 с.
12. Хамахер К., Вранешич З., Заки С. Организация ЭВМ: Пер. с англ., 5-е изд. — СПб.: BHV, Питер, 2003. — 845 с.
13. Цилькер Б. Я., Орлов С. А. Организация ЭВМ и систем. — СПб.: Питер, 2004. — 667 с.
14. Юров В. Assembler. Специальный справочник. — СПб.: Питер, 2000. — 489 с.

Предметный указатель

A

ASCII 40

C

CISC-компьютер 228

E

EBCDIC 41
EPIC-архитектура 231

R

RISC-компьютер 227

T

TLB 192

V

VLIW-архитектура 231

А

Абсолютная погрешность 35, 37
Автодекремент 137
Автоинкремент 138
Адрес:
◊ косвенный 106
◊ непосредственный 108
◊ относительный 112
Адресация:
◊ двойная косвенная 162
◊ сегментная 193
Алгоритм Бута 64
Арифметико-логическое устройство 85

Б

Базирование 168, 169
Байт 105
Байтовая память 105
Блок подготовки перехода 130
Булева операция 47
Булева переменная 47
Булева функция 47
Буфер 178
◊ адресов перехода 223
◊ быстрого преобразования адреса 192

В

Вектор прерывания 172
Вершина стека 139
Выравнивание порядков 69

Д

Дескриптор 180
Дешифрирование кода операции 86
Диспетчер памяти 187
Дополнительный код 28

З

Загрузка регистров 164
Запись:
◊ после записи 221
◊ после чтения 221

И

Иерархическая организация памяти 176
Инструкция *См. Команда*

К

Каталог 180
Код:
◊ ASCII 40
◊ EBCDIC 41
◊ двоично-десятичный 42, 74
Команда 82
◊ безусловного перехода 119
◊ возврата 155
◊ вызова 155
◊ машинная 83
◊ передачи по смещению 124
◊ перехода к подпрограмме 150
◊ с относительной адресацией 123
◊ с переменными адресами 133
◊ стековая 140
◊ условного перехода 119, 131
Компьютер:
◊ с полным набором команд 228
◊ с сокращенным набором команд 227
Константа:
◊ многоразовая 109
◊ одноразовая 109
Контроллер 170, 178
◊ памяти 198
Коррекция:
◊ после сложения 78
◊ при сложении 76
Кэш-память 176, 197

Л

Локальная операция 83
Локальность:
◊ временна́я 177
◊ пространственная 176

М

Мантисса 32
Маска 51
Машинный порядок 33
Многопрограммный режим 182

Н

Настройка по месту 167
Независимость адресных пространств 182
Нормализация 69, 70
◊ вправо 67
Нормализованное число 33

О

Обработчик прерываний 172
Обратная запись 204
Обратный код 27
Оперативная память 80, 81
Операционная система 187
Основные этапы исполнения команды 87
Отложенный переход 221
Относительная ошибка 35
Относительная погрешность 38
Очередь команд 216

П

Память:
◊ виртуальная 183
◊ внешняя 80
◊ последовательного доступа 81
◊ прямого доступа 81
Передача:
◊ по значению 162
◊ по ссылке 162
Переполнение 54, 63
Подпрограмма 149
Порядок:
◊ истинный 66
◊ машинный 66, 71
◊ числа 32
Предсказания переходов 225
Признак:
◊ действительности 180, 201
◊ используемости 180
◊ модификации 180
◊ модифицированности 201
◊ результата 121
Проблема двух нулей 28
Программа 82
Продвижение операндов 219
Прямой код 27

Р

Расслоение:
◊ данных 185
◊ памяти 185
Регистр:
◊ команд 85
◊ общего назначения 100
◊ результата 100

С

Сдвиг:
◊ арифметический 65
◊ влево логический 45
◊ вправо логический 46
Система команд 85
Система счисления 11
◊ *p*-ичная 12
◊ в остаточных классах (СОК) 25
◊ восьмеричная 11
◊ двоичная 18
◊ непозиционная 25
◊ правила перевода 13
◊ пятиричная 23
◊ с неотрицательной базой 23
◊ троичная 23
◊ шестнадцатеричная 12
Сквозная запись 204
Скрытый конфликт по данным 221
Слово 105
Сложение:
◊ беззнаковых чисел 60
◊ в дополнительном коде 57
◊ логическое 48
◊ по модулю 54
◊ по модулю 2 48
◊ с переносом 78
◊ циклическое 55
Слот задержки 221
Смещенный код 29
Сравнение 56
Стек 138
Стековый фрейм 165
Сумматор 92
Суперконвейер 229
Счетчик:
◊ обычный 55
◊ циклический 55

Т

Таблица векторов прерываний 172
Таблица страниц 187
Тег 180
Тело цикла 127

У

Указатель команд 85
Умножение:
◊ логическое 48
◊ целых чисел со знаком 63

Упреждающая выборка 178, 216
Устройства:
◊ ввода 80
◊ внешней памяти 80
◊ вывода 80
◊ управления 89

Ф

Формат:
◊ распакованный 42
◊ упакованный 42

Ц

Центральный процессор 80
Цикл 126
◊ итерационный 129
◊ со счетчиком 128

Ч

Чтение после записи 218

Ш

Шестнадцатеричное число 75

Э

Эффективный адрес 141

Я

Ячейка памяти 81