ОГЛАВЛЕНИЕ

1.4. НЕКОТОРЫЕ ФУНДАМЕНТАЛЬНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

1.4.1. Подпрограммы

Когда некоторую задачу нужно выполнить в нескольких различных местах программы, то, как правило, нежелательно каждый раз повторять ее код. Чтобы избежать этого, данный код (называемый *подпрограммой*) можно поместить только в одном месте и добавить несколько дополнительных команд, чтобы после завершения работы подпрограммы должным образом возобновить выполнение внешней программы. Передача управления между подпрограммами и основной программой называется *связью с подпрограммами*.

Каждый компьютер имеет свои специфические способы установления эффективной связи с подпрограммами, которые обычно подразумевают применение специальных команд. В MIX для этой цели используется регистр J. Рассматривая данную тему, будем ориентироваться на машинный язык MIX, но аналогичные рассуждения применимы и к вопросам связи с подпрограммами для других компьютеров.

Подпрограммы используются с целью экономии места в программе, но их применение не приводит непосредственно к экономии времени. Это происходит неявно вследствие того, что программа занимает меньше места в памяти, например меньше времени тратится на загрузку программы, уменьшается количество проходов в программе либо более эффективно используется высокоскоростная память на машинах с несколькими уровнями памяти. Дополнительным временем, которое тратится на вход в подпрограмму и выход из нее, обычно можно пренебречь.

Подпрограммы имеют и другие преимущества. Благодаря им структура больших и сложных программ становится более наглядной. Они разбивают всю задачу на логические сегменты, что обычно облегчает отладку программы. Многие подпрограммы имеют дополнительную ценность, поскольку ими могут воспользоваться не только авторы, но и другие пользователи.

Большинство компьютерных средств разработки имеют обширные встроенные библиотеки полезных подпрограмм, что значительно облегчает программирование стандартных прикладных задач. Но программист не должен думать, что подпрограммы предназначены только для какой-то *одной* цели. Не следует всегда рассматривать подпрограммы только как программы общего назначения, которыми все могут пользоваться. Не менее важны и подпрограммы специального назначения, даже если они используются только в одной программе. В разделе 1.4.3.1 рассматриваются типичные примеры подпрограмм.

Простейшими являются подпрограммы, которые имеют только один вход и один выход, как, например, подпрограмма MAXIMUM, рассмотренная выше (см. раздел 1.3.2, программа M). Еще раз приведем текст этой программы, изменив ее таким образом, чтобы поиск максимума велся по фиксированному числу ячеек, равному 100.

```
* MAXIMUM OF X[1..100]
MAX100  STJ  EXIT   Связь с подпрограммой.
        ENT3 100    M1. Инициализация.
        JMP  2F
1H      CMPA X,3    M3. Сравнение.
        JGE  *+3
2H      ENT2 0,3    M4. Замена m.
        LDA  X,3    Найден новый максимум.
        DEC3 1      M5. Уменьшение k.
        J3P  1B     M2. Все проверено?
EXIT    JMP  *      Вернуться к основной программе.
```
(1)

В большой программе, содержащей этот код в качестве подпрограммы, с помощью единственной команды "JMP MAX100" можно занести в регистр A значение текущего максимума для ячеек с X + 1 по X + 100 и поместить информацию о положении максимума в rI2. Связь с подпрограммой в этом случае достигается с помощью команд "MAX100 STJ EXIT" и позднее — "EXIT JMP *". Регистр J работает таким образом, что команда выхода затем перейдет в ячейку, следующую за той, из которой было сделано первоначальное обращение к MAX100.

В более новых модификациях компьютеров, таких как машина MMIX, которой суждено заменить MIX, предусмотрены более эффективные способы запоминания адресов возврата. Главное отличие состоит в том, что команды программы больше не модифицируются в памяти; необходимая информация сохраняется в регистрах или в специальном массиве, а не в самой программе (см. упр. 7). В следующем издании данной книги будет применен современный подход к этим вопросам, а пока будем по-прежнему использовать старый самомодифицирующийся код.

Нетрудно получить *количественные* характеристики степени экономичности кода и потери времени при использовании подпрограмм. Предположим, некоторый фрагмент кода занимает k ячеек и встречается в m местах программы. Чтобы оформить этот фрагмент в качестве подпрограммы, понадобится дополнительная команда STJ, строка для команды выхода из подпрограммы плюс по одной команде JMP в каждом из m мест, откуда будет вызываться подпрограмма. В целом, необходимо $m + k + 2$ ячеек, а не mk, поэтому экономия составляет

$$(m - 1)(k - 1) - 3 \tag{2}$$

ячеек. Если k равно 1 либо m равно 1, то, очевидно, мы не сможем сэкономить место в памяти за счет использования подпрограмм. Если k равно 2, то для получения выигрыша m должно быть больше 4, и т. д.

Время теряется из-за применения дополнительных команд JMP, STJ и JMP, которые не присутствуют в программе, если подпрограмма не используется. Поэтому, если во время выполнения основной программы подпрограмма применяется t раз, потребуется $4t$ дополнительных тактов.

К этим оценкам следует подходить с определенной долей скепсиса, так как они делались в расчете на идеальную ситуацию. Многие подпрограммы нельзя вызвать просто с помощью единственной команды JMP. Более того, если фрагмент кода повторяется во многих частях программы и он не оформляется в виде подпрограммы, то для каждой части данный код можно модифицировать так, чтобы получать преимущества от особых характеристик конкретной части программы, в которой он находится. С другой стороны, если принято решение использовать подпрограмму, то ее код нужно писать для наиболее общего, а не частного, случая, и обычно это требует добавления нескольких дополнительных команд.

Если подпрограмма написана для общего случая, то она, как правило, зависит от *параметров*. Параметры — это величины, которые управляют работой подпрограммы; они могут изменяться от одного вызова подпрограммы к другому.

Фрагмент кода внешней программы, который передает управление подпрограмме и должным образом запускает ее, называется *последовательностью вызова*. Конкретные значения параметров, которые передаются при вызове подпрограммы, называются *аргументами*. В нашей подпрограмме MAX100 вызывающая последовательность — это просто команда "JMP MAX100". Но в случае, когда необходимо

передать аргументы, обычно необходима более длинная вызывающая последовательность. Например, программа 1.3.2M — это обобщенный вариант MAX100; она находит максимум среди первых n элементов таблицы. Параметр n появляется в индексном регистре 1, и его последовательность вызова

```
LD1  =n=          или     ENT1  n
JMP  MAXIMUM               JMP   MAXIMUM
```

включает два шага.

Если последовательность вызова занимает c ячеек памяти, то формула (2) вычисления объема сэкономленного места принимает вид

$$(m-1)\,(k-c) - \text{constant}, \tag{3}$$

а время, которое тратится на связь с подпрограммой, немного увеличивается.

Может возникнуть необходимость внесения дальнейших корректив в приведенные формулы, если потребуется сохранить и восстановить некоторые регистры. Например, работая с подпрограммой MAX100, следует помнить, что, записывая "JMP MAX100", мы не только заносим максимальное значение в регистр A, а его позицию — в регистр I2, но и обнуляем регистр I3. Нужно иметь в виду, что подпрограмма может испортить содержимое регистров. Чтобы предотвратить изменение подпрограммой MAX100 содержимого rI3, необходимо включить дополнительные команды. Самый короткий и самый быстрый способ сделать это на MIX состоит в том, чтобы вставить команду "ST3 3F(0:2)" сразу после MAX100 и команду "3H ENT3 *" — непосредственно перед EXIT. В итоге это выльется в две дополнительные строки кода плюс три машинных такта для каждого вызова подпрограммы.

Подпрограмму можно рассматривать как *расширение* машинного языка компьютера. Внеся в память машины подпрограмму MAX100, получим единственную команду (а именно — "JMP MAX100"), которая находит максимум. Важно определить результат работы каждой подпрограммы так же тщательно, как были определены сами операторы машинного языка. Поэтому программист обязательно должен записать характеристики каждой подпрограммы, даже если больше никто не будет пользоваться этой программой или ее спецификацией. Например, подпрограмма MAXIMUM, приведенная в разделе 1.3.2, имеет следующие характеристики.

$$\left.\begin{array}{ll} \text{Последовательность вызова:} & \texttt{JMP MAXIMUM}. \\ \text{Состояние при входе:} & \text{rI1} = n; \quad \text{в предположении, что } n \geq 1. \\ \text{Состояние при выходе:} & \text{rA} = \max\limits_{1\leq k\leq n} \texttt{CONTENTS(X}+k\texttt{)} = \texttt{CONTENTS(X}+\text{rI2}\texttt{)}; \\ & \text{rI3} = 0\text{; содержимое rJ и CI также меняется.} \end{array}\right\} \tag{4}$$

(Обычно мы не будем упоминать о том, что подпрограмма оказывает влияние на регистр J и флаг сравнения; здесь об этом говорится только для полноты.) Заметьте, что на rX и rI1 подпрограмма действия не оказывает, в противном случае эти регистры были бы упомянуты в описании состояния при выходе. В спецификации должны также перечисляться все внешние для подпрограммы ячейки памяти, на которые она может оказать воздействие. В нашем случае спецификация позволяет заключить, что в памяти ничего не сохранялось, так как в (4) ничего не говорится об изменениях в памяти.

А теперь давайте рассмотрим *несколько входов* в подпрограммы. Предположим, существует программа, которой требуется общая подпрограмма MAXIMUM. Но дело в том, что обычно используется частный случай MAX100, для которого $n = 100$. Эти две подпрограммы можно объединить следующим образом.

```
MAX100 ENT3 100    Первый вход.
MAXN   STJ  EXIT   Второй вход.
       JMP  2F     Продолжать, как в (1).                    (5)
...
EXIT   JMP  *      Вернуться к основной программе. ∎
```

Подпрограмма (5) практически такая же, как (1), только первые две команды поменялись местами. Здесь использовался тот факт, что команда "ENT3" не изменяет содержимое регистра J. Чтобы добавить к этой подпрограмме *третий* вход, MAX50, в начало нужно вставить строки

```
MAX50  ENT3 50
       JSJ  MAXN .                                           (6)
```

(Напоминаю, что "JSJ" означает переход без изменения регистра J.)

Если число параметров невелико, то желательно передавать их в подпрограмму одним из двух способов: занеся их в подходящие регистры (аналогично тому, как мы использовали rI3 для хранения параметра n в MAXN, а rI1 — для хранения параметра n в MAXIMUM) либо сохранив их в фиксированных ячейках памяти.

Другой удобный способ передачи аргументов состоит в том, чтобы просто перечислить их *после* команды JMP. Подпрограмма сможет обратиться к своим параметрам, поскольку она знает содержимое регистра J. Например, если бы понадобилось создать для MAXN последовательность вызова

```
       JMP  MAXN
       CON  n   ,                                            (7)
```

то подпрограмму можно было бы написать следующим образом:

```
MAXN   STJ  *+1
       ENT1 *     rI1 ← rJ.
       LD3  0,1   rI3 ← n.
       JMP  2F    Продолжать, как в (1).                     (8)
...
       J3P  1B
       JMP  1,1   Возврат. ∎
```

На таких машинах, как IBM 360, на которых связь с подпрограммами обычно осуществляется путем помещения адреса ячейки возврата в индексный регистр, приведенный выше способ особенно удобен. Он используется также, когда у подпрограммы много аргументов либо если программа создана компилятором. Метод нескольких входов, который использовался выше, в этом случае не подходит. Его можно "подделать", написав

```
MAX100 STJ  1F
       JMP  MAXN
       CON  100
1H     JMP  *   ,
```

но это выглядит уже не так привлекательно, как (5).

Для подпрограмм с *несколькими выходами* обычно используется метод, аналогичный перечислению аргументов. Множественный выход означает, что подпрограмма возвращается в одно из нескольких различных мест в зависимости от условий, обнаруженных подпрограммой. В самом строгом смысле место, в которое возвращается подпрограмма после выхода, — это параметр. Поэтому, если существует несколько мест, в которые она может выйти в зависимости от обстоятельств, они должны быть предоставлены в качестве аргументов. В нашем завершающем примере в подпрограмме поиска максимума будет два входа и два выхода. Последовательность вызова имеет такой вид.

Для произвольного n	Для $n = 100$
`ENT3` n	
`JMP MAXN`	`JMP MAX100`
Выйти здесь,	Выйти здесь,
если $\max \le 0$ или $\max \ge$ rX.	если $\max \le 0$ или $\max \ge$ rX.
Выйти здесь, если $0 < \max <$ rX.	Выйти здесь, если $0 < \max <$ rX.

(Другими словами, выход производится на *две* ячейки ниже команды перехода, если максимум положителен и меньше значения, которое содержится в регистре X.) Для этих условий подпрограмму написать несложно.

```
MAX100 ENT3 100    Вход для n = 100.
MAXN   STJ  EXIT   Вход для произвольного n.
       JMP  2F     Далее, как в (1).
...
       J3P  1B
       JANP EXIT   Выполнить нормальный выход, если max ≤ 0.        (9)
       STX  TEMP
       CMPA TEMP
       JGE  EXIT   Выполнить нормальный выход, если max ≥ rX.
       ENT3 1      В противном случае выйти через второй выход.
EXIT   JMP  *,3    Вернуться в нужное место.  ∎
```

Одни подпрограммы могут вызывать другие подпрограммы. В сложных программах вызовы подпрограмм, вложенных более чем на пять уровней, — это не такое уж редкое явление. Используя описанную выше связь подпрограмм, необходимо придерживаться единственного ограничения — одна подпрограмма не может вызывать другую подпрограмму, которая (прямо или косвенно) вызывает ее. Например, рассмотрим следующий сценарий.

[Главная программа]	[Подпрограмма A]	[Подпрограмма B]	[Подпрограмма C]
	`A STJ EXITA`	`B STJ EXITB`	`C STJ EXITC`
⋮	⋮	⋮	⋮
`JMP A`	`JMP B`	`JMP C`	`JMP A`
⋮	⋮	⋮	⋮
	`EXITA JMP *`	`EXITB JMP *`	`EXITC JMP *` (10)

Если главная программа вызывает `A`, которая вызывает `B`, которая вызывает `C`, а затем `C` вызывает `A`, то адрес в ячейке `EXITA`, по которому осуществляется выход

в главную программу, затирается, что делает возврат к этой программе невозможным. Подобные рассуждения применимы ко всем ячейкам оперативной памяти и к регистрам, используемым подпрограммами. Поэтому разработать правила связи подпрограмм, которые позволят правильно обрабатывать такие рекурсивные ситуации, совсем несложно; подробности приводятся в главе 8.

В завершение этого раздела кратко рассмотрим подходы к написанию сложных и больших программ. Как узнать, какие подпрограммы нам нужны и какие последовательности вызова нужно использовать? Чтобы успешно решить эту задачу, можно воспользоваться методом итераций.

Шаг 0 (Первоначальная идея). Сначала приблизительно выбираем генеральный план действий, которые будут выполняться в программе.

Шаг 1 (Черновая схема программы). Начнем с написания "внешних уровней" программы на любом удобном языке. Систематизированный подход к этому этапу очень хорошо описан в книге E. W. Dijkstra, *Structured Programming* (Academic Press, 1972), Chapter 1, и в работе N. Wirth, *CACM* **14** (1971), 221–227. Начать можно с разбиения всей программы на небольшие фрагменты, которые временно можно рассматривать как подпрограммы, хотя они вызываются только один раз. Эти фрагменты можно последовательно разбивать на все более мелкие части, которые будут служить для выполнения все более простых операций. Каждый раз, когда возникает какая-либо вычислительная задача, которая, похоже, встретится где-то еще либо уже где-то встретилась, следует определить подпрограмму (уже не гипотетическую, а реальную) для выполнения этой задачи. На данном этапе еще не следует писать эту подпрограмму; нужно продолжить написание главной программы, исходя из предположения, что подпрограмма выполнила свою задачу. И наконец, когда схема главной программы будет готова, можно приниматься за подпрограммы, стараясь начинать с самых сложных подпрограмм, постепенно переходя к вложенным в них подпрограммам и т. д. В результате получится список подпрограмм. Функция каждой подпрограммы, вероятно, менялась уже несколько раз, так что к этому моменту начальные элементы схемы будут неправильными. Ничего страшного, ведь это всего лишь схема. Зато теперь мы имеем достаточно четкое представление о том, как будет вызываться каждая подпрограмма и какова степень ее универсальности. Как правило, имеет смысл сделать каждую подпрограмму более универсальной.

Шаг 2 (Первая рабочая программа). Этот шаг по сравнению с шагом 1 — движение в противоположном направлении. Теперь будем использовать компьютерный язык программирования, скажем, `MIXAL`, `PL/MIX` или язык высокого уровня. На этот раз начнем с самых низких уровней вложения подпрограмм, затем "поднимемся вверх" и в конце концов напишем главную программу. Рекомендуется, по возможности, не писать какие-либо команды вызова подпрограммы до того, как будет написан код самой подпрограммы. (На шаге 1 мы старались делать прямо противоположное, не рассматривая подпрограмму до тех пор, пока не будут написаны все ее последовательности вызова.)

По мере написания все большего и большего числа подпрограмм наша уверенность будет постепенно расти, так как мы постоянно расширяем возможности программируемого модуля. После написания отдельной подпрограммы необходимо сразу же подготовить полное описание ее функций и ее последовательностей вызова,

как в (4). Важно также, чтобы не перекрывались ячейки временной памяти. Если каждая подпрограмма будет обращаться к ячейке TEMP, то последствия могут быть катастрофическими, но во время разработки схемы на этапе 1 нам было удобно не беспокоиться об этом. Очевидный способ решения проблем с перекрытием состоит в следующем: нужно выделить для каждой подпрограммы отдельный участок временной памяти, чтобы его использовала только она. Но если такое использование пространства памяти является слишком расточительным, то можно использовать другую схему — присвоить ячейкам имена TEMP1, TEMP2 и т. д. Нумерация внутри подпрограммы начинается с TEMPj, где j на единицу больше, чем наибольший номер, который был использован какой-либо вложенной подпрограммой этой подпрограммы.

Шаг 3 (Повторная проверка). Результатом шага 2 должна быть практически рабочая программа, но вполне возможно, что ее удастся улучшить. Для этого существует хороший метод — снова сменить направление, исследуя для каждой подпрограммы *все* сделанные по отношению к ней вызовы. Может оказаться, что подпрограмму необходимо несколько расширить, чтобы она выполняла распространенные операции, которые всегда осуществляет внешняя программа непосредственно перед использованием подпрограммы или после него. Возможно, несколько подпрограмм следует объединить в одну; а может быть, некоторая подпрограмма вызывается только один раз и из нее вообще не стоит делать подпрограмму. (Случается и такое: подпрограмма не вызывается ни разу и можно вообще обойтись без нее.)

На этом этапе очень часто имеет смысл выбросить все, что было сделано, и снова начать с шага 1! Я вовсе не шучу; время, потраченное на то, чтобы добраться до этого места, не пропало даром, так как вы достаточно глубоко изучили поставленную задачу. Впоследствии (уже после запуска программы), скорее всего, станет ясно, что в общую схему программы необходимо было внести некоторые улучшения. Поэтому нет причин бояться возврата к шагу 1 — намного легче снова пройти шаги 2 и 3 сейчас, когда аналогичная программа уже готова. Скажу еще больше: вполне возможно, что время, которое будет затрачено на переписывание всей программы, с лихвой окупится впоследствии при отладке. Некоторые лучшие из когда-либо написанных компьютерных программ своим успехом во многом обязаны тому, что примерно на этом этапе вся работа была случайно потеряна и авторам пришлось все начать сначала.

С другой стороны, вероятно, в принципе не может наступить момент, когда сложную компьютерную программу нельзя каким-либо образом улучшить. Поэтому шаги 1 и 2 не следует повторять до бесконечности. Когда совершенно очевидно, что можно внести значительное улучшение, имеет смысл потратить дополнительное время на то, чтобы начать все сначала. Но в конце концов наступает стадия “насыщения”, когда сколько-нибудь существенные изменения внести уже нельзя и результатом их воплощения является лишь незначительный прогресс.

Шаг 4 (Отладка). После окончательной “шлифовки” программы, к которой, видимо, относится распределение памяти и другие последние приготовления, самое время посмотреть на нее под еще одним углом зрения, отличным от тех, которые использовались на шагах 1, 2 и 3. Теперь мы будем исследовать элементы программы в том порядке, в котором их будет *выполнять* компьютер. Это можно сделать вручную или, конечно, с помощью компьютера. Автор пришел к выводу,

что на данном этапе очень полезно использовать системные программы, которые прослеживают каждую команду, когда она выполняется первые два раза. Очень важно заново продумать идеи, лежащие в основе программы, и убедиться в том, что все действительно происходит так, как ожидалось.

Отладка — это искусство, которое требует дальнейшего изучения, и способ ее проведения в значительной степени зависит от имеющихся на компьютере устройств. Хорошим началом эффективной отладки во многих случаях может стать подготовка соответствующих тестовых данных. Самыми эффективными, похоже, являются те методы отладки, которые предназначены для конкретной программы и встроены именно в нее. Многие из лучших программистов современности почти половину своих программ посвятили тому, чтобы облегчить отладку программ из другой половины. "Первая половина", которая обычно состоит из достаточно простых программ, отображающих соответствующую информацию в удобном для чтения виде, в конце концов выбрасывается, но в итоге это дает удивительный выигрыш в производительности.

Еще один хороший метод отладки состоит в том, чтобы вести учет всех сделанных ошибок. Конечно, это нелегко, но подобная информация является бесценной для каждого, кто пытается отладить программу; она также поможет в дальнейшем избежать множества ошибок.

Замечание. Большинство предыдущих комментариев было написано автором в 1964 году, после того как он успешно завершил несколько программных проектов среднего масштаба, но до того как он приобрел зрелый стиль программирования. Впоследствии, в 80-х годах, он понял, что, вероятно, еще более важным является метод, который называется *структурным документированием* или *грамотным программированием*. Анализ современных представлений об оптимальных способах написания всевозможных программ содержится в книге *Literate Programming* (Cambridge Univ. Press), впервые опубликованной в 1992 году. Кстати, в главе 11 этой книги приведен подробный перечень всех ошибок, которые были устранены из программы TEX в период между 1978 и 1991 годами.

До некоторого момента лучше допустить наличие ошибок в программе, чем потратить на ее разработку столько времени, сколько необходимо для устранения всех ошибок (сколько десятилетий на это потребуется?).

— А. М. ТЬЮРИНГ, *Proposals for ACE* (1945)

УПРАЖНЕНИЯ

1. [*10*] Сформулируйте характеристики подпрограммы (5). В качестве образца используйте (4), где даются характеристики подпрограммы 1.3.2M.

2. [*10*] Предложите код, заменяющий (6). Не используйте команду JSJ.

3. [*M15*] Дополните информацию, которая дана в (4), точно указав, что происходит с регистром J и флагом сравнения при выполнении подпрограммы. Определите также, что происходит в случае, когда в регистре I1 содержится неположительное число.

▸ **4.** [*21*] Напишите обобщающую MAXN подпрограмму нахождения максимального значения для последовательности X[a], X[$a+r$], X[$a+2r$], ..., X[n], где r и n — параметры, а a — наименьшее положительное число, для которого $a \equiv n$ (по модулю r), т. е.

$a = 1 + (n - 1) \bmod r$. Предусмотрите специальный вход для случая $r = 1$. Перечислите характеристики новой подпрограммы, взяв за образец (4).

5. [*21*] Предположим, в машине MIX нет регистра J. Придумайте способ связи подпрограмм, не использующий регистр J. Проиллюстрируйте свое изобретение на примере, написав подпрограмму MAX100, фактически эквивалентную (1). Сформулируйте характеристики этой подпрограммы аналогично тому, как это сделано в (4). (Придерживайтесь принятых для MIX соглашений о самомодифицирующемся коде.)

▸ **6.** [*26*] Предположим, в MIX нет оператора MOVE. Напишите подпрограмму с именем MOVE, такую, чтобы ее последовательность вызова "JMP MOVE; NOP A,I(F)" давала такой же эффект, как и "MOVE A,I(F)", если бы последнее было допустимо. Единственные отличия должны заключаться в воздействии на регистр J и в том, что время выполнения подпрограммы несколько увеличится.

▸ **7.** [*20*] Почему к самомодифицирующемуся коду сейчас относятся неодобрительно?

1.4.2. Сопрограммы

Подпрограммы — это частные случаи более общих программных компонентов, называемых *сопрограммами*. В противоположность несимметричной связи между главной программой и подпрограммой, между сопрограммами, которые *вызывают одна другую*, существует полная симметрия.

Чтобы понять, что представляет собой сопрограмма, давайте посмотрим на подпрограммы с другой стороны. В предыдущем разделе мы придерживались той точки зрения, что подпрограмма — это некая аппаратная реализация, которая используется для сокращения количества строк в программе. Это, конечно, правильно, но возможна и другая точка зрения. Можно рассматривать главную программу и подпрограмму как *группу* программ, причем каждый член группы должен выполнять определенную работу. Главная программа в процессе своей работы активизирует подпрограмму, последняя выполняет собственную функцию, а затем активизирует главную программу. Мы можем выйти за узкие рамки своих представлений и вообразить, что с точки зрения подпрограммы, когда она осуществляет выход, именно *она* вызывает *главную* программу. Главная программа продолжает выполнять свои обязанности, а затем "выходит" в подпрограмму. Подпрограмма работает, а затем снова вызывает главную программу.

Эта несколько надуманная философия на самом деле полностью отражает ситуацию с сопрограммами, для которых невозможно определить, где главная программа, а где подпрограмма. Предположим, имеются сопрограммы A и B. Программируя A, можно считать, что B — это подпрограмма; в свою очередь, программируя B, как подпрограмму можно рассматривать уже A. Таким образом, в сопрограмме A команда "JMP B" используется для активизации сопрограммы B. В сопрограмме B команда "JMP A" применяется для того, чтобы снова активизировать сопрограмму A. Каждый раз при активизации сопрограмма возобновляет выполнение своей программы с той точки, в которой действие было приостановлено в прошлый раз.

Сопрограммами A и B могут быть, например, две программы, играющие в шахматы. Их можно скомбинировать так, чтобы они играли одна против другой.

В MIX подобная связь между сопрограммами A и B осуществляется путем включения в программу следующих четырех команд.

```
A   STJ  BX          B   STJ  AX
AX  JMP  A1          BX  JMP  B1
```
(1)

Для передачи управления любой из сопрограмм требуется четыре машинных такта. Первоначально AX и BX установлены на переход к начальным точкам каждой сопрограммы, A1 и B1. Предположим, сначала запускается сопрограмма A; она начинает выполняться с команды, которая находится в ячейке A1. Когда сопрограмма A выполняет, скажем, команду "JMP B" из ячейки A2, команда из ячейки B сохраняет в AX содержимое rJ, скажем, команду "JMP A2+1". Команда из BX переносит нас в ячейку B1, и вслед за этим начинает выполняться сопрограмма B. В конце концов она доходит до команды "JMP A", которая находится, скажем, в ячейке B2. Мы сохраняем содержимое rJ в BX и переходим к ячейке A2+1, продолжая выполнять сопрограмму A до тех пор, пока управление снова не перейдет к B, которая сохраняет содержимое регистра J в AX, переходит к B2+1 и т. д.

Главное различие между связями "программа — подпрограмма" и "сопрограмма — сопрограмма", как видно из предыдущего примера, состоит в том, что подпрограмма всегда начинается *с самого начала* (как правило, это фиксированная точка), а главная программа или сопрограмма всегда начинается *с места, следующего* за той точкой, в которой ее выполнение было прекращено в прошлый раз.

Необходимость использования сопрограмм на практике естественным образом возникает в случае, когда они связаны с алгоритмами ввода и вывода. Приведем такой пример. Предположим, что в обязанности сопрограммы A входит считывание перфокарт и выполнение такого преобразования входных данных, которое сводит их к последовательности элементов. Другая сопрограмма, которую мы будем называть B, выполняет дальнейшую обработку этих элементов и печатает ответы. B периодически запрашивает последующие элементы ввода, полученные A. Таким образом, сопрограмма B вызывает A каждый раз, когда ей нужен следующий элемент ввода, и сопрограмма A вызывает B каждый раз, когда находит элемент ввода. Читатель может сказать: "Ну что ж, B — это главная программа, а A — просто *подпрограмма* для выполнения ввода". Но это утверждение становится уже не таким правильным, когда процесс A оказывается очень сложным. В действительности можно считать, что A — это главная программа, а B — подпрограмма, выполняющая вывод, и приведенное выше описание останется верным. Но полезность идеи сопрограммы обнаруживается между этими двумя крайностями, когда и A, и B достаточно сложны и каждая из них вызывает другую много раз из различных точек. Подобрать небольшие, простые примеры сопрограмм, иллюстрирующие важность этой идеи, весьма трудно. В случаях, когда применение сопрограмм приносит наибольшую пользу, эти сопрограммы, как правило, оказываются довольно большими.

Чтобы рассмотреть сопрограммы в действии, воспользуемся таким примером. Предположим, необходимо написать программу, преобразующую один код в другой. Входной код, который нужно преобразовать, представляет собой последовательность буквенно-цифровых символов, заканчивающуюся точкой, например

```
A2B5E3426FGOZYW3210PQ89R.
```
(2)

Эта последовательность выбивается на перфокартах; пустые колонки, встречающиеся на перфокартах, игнорируются. Входные данные интерпретируются следующим образом, слева направо: если следующий символ — цифра `0`, `1`, ..., `9` (обозначим ее через n), то он указывает на $(n + 1)$ повторение следующего символа независимо от того, является ли он цифрой. Нецифровой символ обозначает сам себя. Выходные данные нашей программы должны представлять собой последовательность, полученную указанным образом и разделенную на группы по три символа. Последовательность заканчивается с появлением точки; в последней группе может быть менее трех символов. Например, последовательность (2) должна быть преобразована нашей программой в следующие группы символов:

```
ABB BEE EEE E44 446 66F GZY W22 220 0PQ 999 999 999 R.        (3)
```

Обратите внимание, что `3426F` означает не 3427 повторений буквы `F`, а четыре четверки и три шестерки, за которыми следует буква `F`. Если входная последовательность имеет вид '`1.`', то на выходе будет просто '`.`', а не '`..`', поскольку первая же точка заканчивает вывод. Наша программа должна выбить выходные данные на перфокартах — по шестнадцать групп на каждой карте, за исключением, возможно, последней.

Для выполнения такого преобразования напишем две сопрограммы и одну подпрограмму. Подпрограмма с именем `NEXTCHAR` предназначена для нахождения непустых символов (т. е. не пробелов) во входных данных и помещения каждого последующего такого символа в регистр A.

```
01  * ПОДПРОГРАММА ВВОДА СИМВОЛОВ
02  READER   EQU  16              Номер устройства чтения перфокарт.
03  INPUT    ORIG *+16            Место для входных карт.
04  NEXTCHAR STJ  9F              Вход в подпрограмму.
05           JXNZ 3F              Первоначально rX = 0.
06  1H       J6N  2F              Первоначально rI6 = 0.
07           IN   INPUT(READER)   Читать следующую карту.
08           JBUS *(READER)       Ожидать завершения.
09           ENN6 16              Пусть rI6 указывает на следующее слово.
10  2H       LDX  INPUT+16,6      Взять следующее слово из ввода.
11           INC6 1               Продвинуть указатель.
12  3H       ENTA 0
13           SLAX 1               Следующий символ → rA.
14  9H       JANZ *               Пропустить пробелы.
15           JMP  NEXTCHAR+1      ∎
```

Эта подпрограмма имеет следующие характеристики.

Последовательность вызова: `JMP NEXTCHAR`.
Состояние при входе: rX = количество символов, которые еще будут использованы; rI6 указывает на следующее слово или rI6 = 0, указывая, что нужно читать новую карту.
Состояние при выходе: rA = следующий непустой символ ввода; rX и rI6 настроены для следующего входа в `NEXTCHAR`.

Наша первая сопрограмма с именем IN находит символы входной последовательности и количество их повторений. В первый раз ее выполнение начинается с IN1.

```
16  * ПЕРВАЯ СОПРОГРАММА
17  2H      INCA 30          Найден нецифровой символ.
18          JMP  OUT         Отослать его в сопрограмму OUT.
19  IN1     JMP  NEXTCHAR    Получить символ.
20          DECA 30
21          JAN  2B          Это буква?
22          CMPA =10=
23          JGE  2B          Это специальный символ?
24          STA  *+1(0:2)    Найдена цифра n.
25          ENT5 *           rI5 ← n.
26          JMP  NEXTCHAR    Взять следующий символ.
27          JMP  OUT         Отослать его в сопрограмму OUT.
28          DEC5 1           Уменьшить n на 1.
29          J5NN *-2         Повторить в случае необходимости.
30          JMP  IN1         Начать новый цикл. ∎
```

(Напоминаю, что в символьном коде MIX цифры 0–9 имеют коды 30–39.) Эта сопрограмма имеет следующие характеристики.

Последовательность вызова: JMP IN.
Состояние при выходе (при вызове OUT): rA = следующий символ ввода с соответствующим числом повторений; содержимое rI4 остается таким же, как при входе.
Состояние при входе (после возврата): содержимое rA, rX, rI5, rI6 должно оставаться неизменным с момента последнего выхода.

Вторая сопрограмма с именем OUT разбивает код на группы по три символа и осуществляет перфорацию. Ее выполнение начинается с OUT1.

```
31  * ВТОРАЯ СОПРОГРАММА
32          ALF                      Постоянная, используемая для пробелов.
33  OUTPUT  ORIG *+16                Буферная область для ответов.
34  PUNCH   EQU  17                  Номер устройства перфорирования.
35  OUT1    ENT4 -16                 Начать новую выходную перфокарту.
36          ENT1 OUTPUT
37          MOVE -1,1(16)            Занести в область вывода пробелы.
38  1H      JMP  IN                  Взять следующий преобразованный символ.
39          STA  OUTPUT+16,4(1:1)    Сохранить его в поле (1:1).
40          CMPA PERIOD              Это "."?
41          JE   9F
42          JMP  IN                  Если нет, взять другой символ.
43          STA  OUTPUT+16,4(2:2)    Сохранить его в поле (2:2).
44          CMPA PERIOD              Это "."?
```

```
45          JE   9F
46          JMP  IN                  Если нет, взять следующий символ.
47          STA  OUTPUT+16,4(3:3)    Сохранить его в поле (3:3).
48          CMPA PERIOD              Это "."?
49          JE   9F
50          INC4 1                   Перейти к следующему слову в буфере вывода.
51          J4N  1B                  Конец карты?
52  9H      OUT  OUTPUT(PUNCH)       Если да, перфорировать ее.
53          JBUS *(PUNCH)            Ожидать завершения.
54          JNE  OUT1                Вернуться за следующими символами, если
55          HLT                        не появилась ".".
56  PERIOD  ALF  ␣␣␣␣.                 ∎
```

Эта сопрограмма имеет следующие характеристики.

Последовательность вызова: `JMP OUT`.
Состояние при выходе (при вызове `IN`): Содержимое регистров rA, rX, rI5, rI6 остается неизменным с момента входа; значение в rI1 может измениться; предыдущий символ записывается в выходные данные.
Состояние при входе (при возврате): rA = следующий символ ввода с числом повторений; значение в rI4 не меняется с момента последнего выхода.

Для завершения программы нужно обеспечить связь между сопрограммами (см. (1)) и должным образом выполнить инициализацию. Инициализация сопрограмм — это довольно тонкое, хотя и несложное, дело.

```
57  * ИНИЦИАЛИЗАЦИЯ И СВЯЗЬ
58  START  ENT6 0       Инициализировать rI6 для NEXTCHAR.
59         ENTX 0       Инициализировать rX для NEXTCHAR.
60         JMP  OUT1    Начать с OUT (см. упр. 2).
61  OUT    STJ  INX     Связь сопрограмм.
62  OUTX   JMP  OUT1
63  IN     STJ  OUTX
64  INX    JMP  IN1
65         END  START   ∎
```

Теперь программа полностью готова. Читателю следует тщательно ее изучить, в особенности обращая внимание на то, как можно независимо написать каждую сопрограмму, считая, что другая сопрограмма — это ее подпрограмма.

В приведенной выше программе состояния при входе и при выходе для сопрограмм `IN` и `OUT` идеально согласованы. Но в более общем случае нам вряд ли так повезет и, чтобы связать сопрограммы, придется также включить команды загрузки и сохранения соответствующих регистров. Например, если бы программа `OUT` изменяла содержимое регистра A, то связь сопрограмм нужно было бы запро-

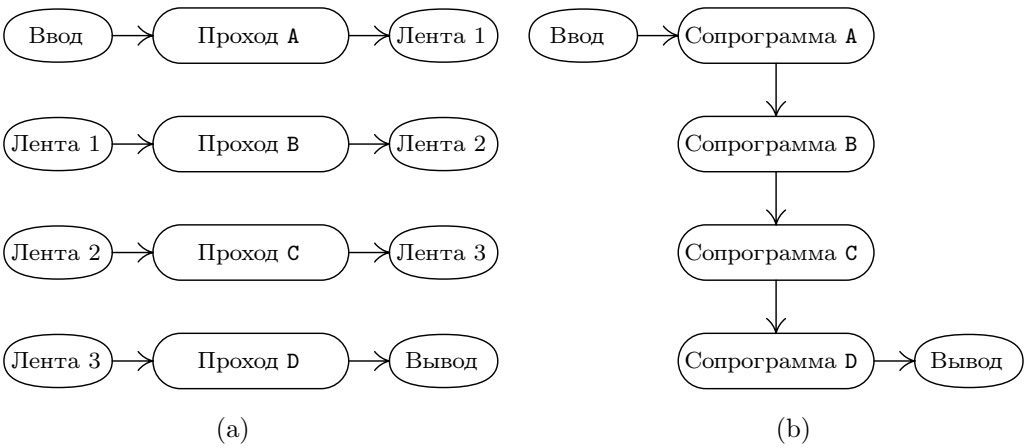


Рис. 22. Проходы: (a) четырехпроходный алгоритм и (b) однопроходный алгоритм.

граммировать следующим образом.

```
OUT  STJ  INX
     STA  HOLDA  Сохранить A при выходе из IN.
OUTX JMP  OUT1
IN   STJ  OUTX                                    (4)
     LDA  HOLDA  Восстановить A при выходе из OUT.
INX  JMP  IN1
```

Существует важная связь между сопрограммами и *многопроходными алгоритмами*. Например, описанный выше процесс преобразования можно выполнить за два отдельных прохода. Можно сначала выполнить только сопрограмму `IN`, применяя ее ко всему вводу и записывая каждый символ с соответствующим числом повторений на магнитную ленту, а затем перемотать ленту в начало и выполнить только сопрограмму `OUT`, выбирая символы с ленты группами по три. Такой процесс называется двухпроходным. (На интуитивном уровне термин "проход" воспринимается как полный просмотр входных данных. Это определение является неточным, и во многих алгоритмах совсем не очевидно, чему равно число выполненных проходов. Но, несмотря на некоторую неопределенность данного термина, полезно понять его на интуитивном уровне.)

На рис. 22, (a) показан четырехпроходный процесс. Во многих случаях будет оказываться, что такой же процесс можно выполнить только за один проход, как показано в части (b) рисунка, если заменить сопрограммами `A`, `B`, `C`, `D` соответствующие проходы `A`, `B`, `C`, `D`. Сопрограмма `A` вызывает `B`, после того как во время прохода `A` элемент выходных данных записан на ленту 1. Сопрограмма `B` вызывает `A`, после того как во время прохода `B` элемент входных данных считан с ленты 1. Сопрограмма `B` вызывает `C`, после того как во время прохода `B` элемент выходных данных записан на ленту 2 и т. д. Пользователи UNIX® узнают в этом "канал", который обозначается как `ПроходA | ПроходB | ПроходC | ПроходD`. Программы, соответствующие проходам `B`, `C` и `D`, иногда называются фильтрами.

И наоборот, процесс, выполняемый n сопрограммами, часто можно преобразовать в n-проходный процесс. Именно из-за этого соответствия имеет смысл провести сравнительный анализ многопроходных и однопроходных алгоритмов.

a) *Психологическое отличие.* Для одной и той же задачи, как правило, проще создать и понять многопроходный алгоритм, чем однопроходный. Процесс, разделенный на ряд небольших шагов, выполняющихся один после другого, понять намного легче, чем запутанную процедуру, в которой множество преобразований выполняется одновременно.

Кроме того, если нужно написать большую серьезную программу, над которой будет совместно работать многочисленная группа разработчиков, то многопроходный алгоритм поможет естественным путем распределить работу между участниками проекта.

Но такие преимущества многопроходного алгоритма присущи и сопрограммам, так как каждую из них можно написать, по существу, отдельно от других, а связь между сопрограммами превратит явно многопроходный алгоритм в однопроходный процесс.

b) *Разница во времени выполнения.* В однопроходном алгоритме не нужно тратить время на то, чтобы упаковать, записать, прочитать и распаковать промежуточные данные, которые передаются между проходами (например, информация на лентах на рис. 22). Поэтому однопроходный алгоритм выполняется быстрее.

c) *Различие в объеме памяти.* Для однопроходного алгоритма необходимо хранить в памяти все программы одновременно, в то время как для многопроходного алгоритма можно сохранять в памяти только по одной программе. Этот фактор может отразиться на скорости выполнения программы даже в большей степени, чем фактор, указанный в п. (b). Например, многие компьютеры имеют ограниченный объем "быстродействующей памяти" и большой объем "медленной памяти". Если каждый проход поместится в быстрой памяти, то вся программа выполнится значительно быстрее, чем в случае использования сопрограмм в одном проходе (так как при использовании сопрограмм большинство программ, скорее всего, будут помещены в "медленную память" либо будут много раз загружаться и выгружаться из быстрой памяти).

Иногда возникает необходимость в разработке алгоритмов сразу для нескольких компьютерных конфигураций, одни из которых имеют больший объем памяти, чем другие. В таких случаях можно написать программу в виде набора сопрограмм и поставить число проходов в зависимость от размера памяти: загрузить сразу столько сопрограмм, сколько поместится, а подачу данных по недостающим связям обеспечить с помощью подпрограмм ввода или вывода.

Несмотря на важность связи между сопрограммами и проходами, нужно иметь в виду, что программу, написанную в виде набора сопрограмм, не всегда можно представить в виде многопроходного алгоритма. Если сопрограмма `B` получает входные данные от `A` и отправляет обратно ключевую информацию сопрограмме `A` (как в приведенном выше примере программы игры в шахматы), то последовательность таких действий нельзя преобразовать в проход `A`, за которым следует проход `B`.

И наоборот, ясно, что некоторые многопроходные алгоритмы нельзя преобразовать в сопрограммы. Отдельные алгоритмы являются многопроходными по своей

сути; например, для второго прохода может требоваться совокупная информация от первого прохода (скажем, общее число появлений некоторого слова во вводе). В этом отношении показательна следующая старая шутка.

> *В автобусе. Маленькая старушка:* "Мальчик, ты не подскажешь мне, когда нужно выходить, чтобы попасть на Пасадена-Стрит?".
>
> *Мальчик:* "Просто следите за мной и выходите за две остановки перед тем, как выйду я".

(Шутка заключается в том, что мальчик предлагает старушке двухпроходный алгоритм.)

Вот пока и все, что касается многопроходных алгоритмов. С другими примерами сопрограмм мы будем встречаться в различных разделах книги, например они будут частью буферных схем в разделе 1.4.4. Сопрограммы играют также важную роль в моделировании дискретных систем (см. раздел 2.2.5). Важная идея *репликационных сопрограмм* обсуждается в главе 8, а некоторые интересные примеры применения этой идеи можно найти в главе 10.

УПРАЖНЕНИЯ

1. [*10*] Объясните, почему автору трудно найти небольшие простые примеры сопрограмм.

▸ **2.** [*20*] В программе, приведенной в тексте раздела, первой запускается сопрограмма `OUT`. Что произойдет, если первой будет выполняться сопрограмма `IN`, т. е. если в строке 60 поменять "`JMP OUT1`" на "`JMP IN1`"?

3. [*20*] Истинно ли утверждение "Все три команды "`CMPA PERIOD`" из программы `OUT` можно опустить, и программа по-прежнему будет работать"? (Будьте внимательны.)

4. [*20*] Покажите, как связь между сопрограммами, аналогичную (1), можно реализовать на реальных компьютерах, которые вы хорошо знаете.

5. [*15*] Предположим, для обеих сопрограмм `IN` и `OUT` нужно, чтобы в промежутке между входом и выходом содержимое регистра A оставалось неизменным. Другими словами, предположим, что в каждом случае появления команды "`JMP IN`" внутри сопрограммы `OUT` содержимое регистра A должно оставаться неизменным до возврата управления следующей строке. Аналогичное предположение делается по отношению к команде "`JMP OUT`" внутри сопрограммы `IN`. Как в этом случае должна выглядеть связь между сопрограммами? (Ср. с (4).)

▸ **6.** [*22*] Напишите команды связи между сопрограммами, аналогичные (1), для случая *трех* сопрограмм, `A`, `B` и `C`, каждая из которых может вызывать любую из двух других. (В каждом случае активизации сопрограммы выполнение начинается с того места, где оно закончилось в прошлый раз.)

▸ **7.** [*30*] Напишите программу для `MIX`, которая выполняет преобразование, *обратное* тому, которое осуществляет программа из текста раздела, т. е. ваша программа должна получать на входе перфокарты типа (3), а на выходе выдавать перфокарты типа (2). На выходе должна быть настолько короткая строка символов, насколько это возможно, поэтому нуль перед буквой `Z` в (2) на этот раз не должен быть перенесен из (3).

1.4.3. Программы-интерпретаторы

В этом разделе будет рассмотрен один из распространенных типов компьютерных программ — *программы-интерпретаторы* (которые для краткости называют просто *интерпретаторами*). Интерпретатор — это компьютерная программа, которая выполняет команды другой программы, написанной на некотором машинно-ориентированном языке. Под машинно-ориентированным языком подразумевается такой способ представления команд, который предполагает использование в командах кодов операций, адресов и т. д. (Это определение, как и большинство определений современных компьютерных терминов, не является точным, да оно и не должно быть таким; нельзя однозначно определить, какие программы являются интерпретаторами, а какие — нет.)

Исторически первые интерпретаторы являлись оболочками машинно-ориентированных языков, специально предназначенными для упрощения процесса программирования. Такими языками было гораздо легче пользоваться, чем настоящими машинными языками. Возникновение символических языков программирования привело к тому, что отпала необходимость в программах-интерпретаторах такого типа, но это никоим образом не означает, что интерпретаторы начали постепенно исчезать. Наоборот, их продолжали применять и теперь применяют настолько широко, что эффективное использование программ-интерпретаторов можно считать одной из главных характеристик современного программирования. Новые сферы применения интерпретаторов появились, в основном, по следующим причинам:

a) на машинно-ориентированном языке можно представить сложную последовательность решений и действий в компактном и удобном виде;

b) такое представление обеспечивает прекрасный способ передачи информации между проходами в многопроходном процессе.

В подобных случаях разрабатываются машинно-ориентированные языки специального назначения для использования в конкретной программе и программы, написанные на этих языках, часто генерируются только компьютерами. (Современные квалифицированные программисты являются также хорошими разработчиками машин, поскольку они не только создают программу-интерпретатор, но и определяют *виртуальную машину*, язык которой необходимо интерпретировать.)

Принцип интерпретирования имеет еще одно дополнительное преимущество, которое состоит в относительной независимости от машины. Это означает, что при переходе от одного компьютера к другому необходимо переписать только интерпретатор. Более того, полезные средства отладки можно легко встроить в интерпретирующую систему.

Примеры интерпретаторов типа (a) приводятся ниже в нескольких разделах данного многотомника (например, рекурсивный интерпретатор — в главе 8 и машина для синтаксического анализа — в главе 10). Во многих ситуациях, как правило, приходится иметь дело с большим числом аналогичных задач, для которых не удается придумать описывающую их простую общую схему.

Рассмотрим такой пример. Предположим, необходимо написать алгебраический компилятор, с помощью которого можно генерировать эффективные команды машинного языка для сложения двух величин. Эти величины могут принадлежать одному из десяти классов (константы, простые переменные, рабочие ячейки, ин-

дексные переменные, содержимое аккумулятора или индексного регистра, числа с фиксированной или плавающей точкой и т. д.). Если подсчитать количество комбинаций всех пар, то получится 100 различных случаев. И для того чтобы в каждом случае правильно выполнить операцию, понадобится длинная программа. Для решения данной задачи методом интерпретирования нужно изобрести специальный язык, "команды" которого помещались бы в одном байте. Затем необходимо просто подготовить таблицу из 100 "программ" на этом языке, таких, чтобы каждая программа идеально помещалась в одном слове. Идея состоит в том, чтобы выбирать из таблицы соответствующий элемент-программу и выполнять ее. Это простой и эффективный метод.

Пример интерпретатора типа (b) приведен в статье Д. Э. Кнута (D. E. Knuth), "Computer-Drawn Flowcharts", *CACM* **6** (1963), 555–563. В многопроходной программе предыдущие проходы должны передавать информацию последующим. Наиболее эффективным средством передачи этой информации следующему проходу является набор команд на машинно-ориентированном языке. Тогда последующий проход — это ни что иное, как программа-интерпретатор специального назначения, а предыдущий проход — это "компилятор" специального назначения. Такую философию многопроходного процесса можно охарактеризовать следующим образом: мы, по возможности, *рассказываем* последующему проходу, что нужно делать, а не просто передаем ему набор фактов и просим *понять*, что необходимо сделать.

Другой пример интерпретатора типа (b) связан с компиляторами для специальных языков. Если в язык включено много возможностей, которые достаточно просто можно реализовать только в виде подпрограмм, то полученные в результате объектные программы будут представлять собой очень длинные последовательности вызовов подпрограмм. Это может случиться, например, если язык предназначен, главным образом, для выполнения арифметических действий с высокой точностью. В подобном случае объектная программа будет значительно короче, если ее написать на интерпретируемом языке. Например, в книге B. Randell, L. J. Russell, *ALGOL 60 Implementation* (New York: Academic Press, 1964) описывается компилятор, выполняющий трансляцию с языка ALGOL 60 на интерпретируемый язык, а также рассказывается об интерпретаторе для этого языка. В работе Arthur Evans, Jr., "An ALGOL 60 Compiler", *Ann. Rev. Auto. Programming* **4** (1964), 87–124, приводятся также примеры программ-интерпретаторов, которые используются *во внутренней* структуре компилятора. Повсеместное распространение микрокомпьютеров и интегральных микросхем специального назначения сделало этот метод интерпретирования еще более ценным.

Написанная на TEX программа, с помощью которой были созданы страницы этой книги, преобразовала файл, содержащий текст настоящего раздела, в интерпретируемый язык. Этот язык, который называется форматом `DVI`, был разработан Д. Р. Фучсом (D. R. Fuchs) в 1979 году. [См. D. E. Knuth, *TEX: The Program* (Reading, Mass.: Addison-Wesley, 1986), Part 31.] `DVI`-файл, созданный TEX, затем был обработан интерпретатором `dvips`, который написал Т. Г. Рокики (T. G. Rokicki), и преобразован в файл команд на другом интерпретируемом языке под названием PostScript® [Adobe Systems Inc., *PostScript Language Reference Manual*, 2nd edition (Reading, Mass.: Addison-Wesley, 1990)]. Этот PostScript-файл был отослан в издательство, где его распечатали на фотонаборной машине, в которой для получения

печатных пластин используется PostScript-интерпретатор. Такая трехпроходная операция является наглядной иллюстрацией интерпретаторов типа (b); в сам TEX также включен небольшой интерпретатор типа (a), предназначенный для обработки так называемой лигатуры и выполнения кернинга для символов каждого шрифта, встречающегося в тексте [*TEX: The Program*, §545].

На программу, написанную на интерпретируемом языке, можно посмотреть и с другой точки зрения. Ее можно считать набором вызовов подпрограмм, следующих один за другим. Подобную программу можно легко развернуть в длинную последовательность вызовов подпрограмм, и наоборот: такую последовательность обычно можно свернуть в набор команд, который легко интерпретируется. Итак, к преимуществам методов интерпретирования относятся компактность представления, машинная независимость и расширенные возможности диагностики. Во многих случаях интерпретатор можно написать так, чтобы затраты времени на интерпретацию самого кода и на переход к нужной программе были незначительны.

1.4.3.1. Имитатор MIX. Если язык, который должен обрабатываться интерпретатором, является машинным языком другого компьютера, то этот интерпретатор обычно называют *имитатором* (а иногда *эмулятором*).

По мнению автора, на написание таких имитаторов потрачено слишком много времени работы программистов, а на их использование — слишком много компьютерного времени. Мотивы создания имитаторов очень просты. Например, начальник компьютерного отдела покупает новую машину и хочет по-прежнему использовать на ней программы, написанные для старой машины (вместо того чтобы переписать их с учетом особенностей новой машины). Но обычно это стоит дороже и дает худшие результаты, чем в случае, когда временно нанимается группа программистов и перед ней ставится задача выполнить перепрограммирование. Например, однажды автор принимал участие в подобном проекте и в первоначальной программе, которая использовалась в течение нескольких лет, была обнаружена серьезная ошибка. Мало того что новая программа давала правильные результаты, она еще и работала в пять раз быстрее старой! (Не все имитаторы плохи. Например, для компьютерной фирмы-производителя обычно очень полезно сымитировать новую машину еще до того, как она будет запущена в производство, чтобы программное обеспечение для нее можно было разработать как можно скорее. Но это очень узкая область применения имитаторов.) В качестве яркого примера неэффективного использования имитаторов компьютеров можно привести подлинную историю о машине A, имитирующей машину B, на которой работает программа, имитирующая машину C! Данный способ приводит к тому, что большой и дорогой компьютер дает худшие результаты по сравнению со своим более дешевым собратом.

Ввиду всего вышесказанного возникает вопрос, почему же этот имитатор "поднял свою уродливую голову" в данной книге? На это есть две причины.

a) Имитатор, который будет описан ниже, — это хороший пример типичной программы-интерпретатора. Здесь проиллюстрированы основные методы, использующиеся в интерпретаторах, и, кроме того, демонстрируется применение подпрограмм в достаточно длинной программе.

b) Будет рассмотрен имитатор компьютера MIX, написанный на языке MIX (подумать только!). Это облегчит написание имитаторов MIX для большинства компью-

теров, подобных MIX; в коде нашей программы мы намеренно избегали широкого использования возможностей, присущих исключительно MIX. Имитатор MIX пригодится вам в качестве наглядного пособия к этой книге и, возможно, к другим.

Компьютерные имитаторы, описываемые в настоящем разделе, следует отличать от *имитаторов дискретных систем*. Имитаторы дискретных систем — это важные программы, которые будут обсуждаться в разделе 2.2.5.

А теперь вернемся к задаче написания имитатора MIX. Входными данными для нашей программы будут последовательность команд MIX и данные, сохраненные в ячейках 0000–3499. Мы в точности сымитируем работу аппаратного обеспечения MIX и сделаем вид, что MIX сам интерпретирует эти команды. Таким образом мы попытаемся реализовать спецификации, которые были определены в разделе 1.3.1. В нашей программе, например, используется переменная AREG, с помощью которой сохраняется модуль значения, содержащегося в имитируемом регистре A; другая переменная, SIGNA, используется для хранения соответствующего знака. С помощью переменной CLOCK ведется учет количества единиц имитируемого времени MIX, затраченного на выполнение имитируемой программы.

Нумерация таких команд MIX, как LDA, LD1, ..., LDX и других подобных команд, подсказывает нам, что сохранять имитируемое содержимое этих регистров в последовательных ячейках нужно так:

AREG, I1REG, I2REG, I3REG, I4REG, I5REG, I6REG, XREG, JREG, ZERO.

Здесь ZERO — это "регистр", постоянно заполненный нулями. Позиции JREG и ZERO выбраны в соответствии с кодами операций команд STJ и STZ.

Согласно нашей философии написания имитатора, т. е. ориентируясь на любой компьютер, а не только на MIX, будем рассматривать знаки как независимые части регистра. Например, во многих компьютерах нельзя представить число "минус нуль", а в MIX можно, поэтому в данной программе знаки всегда будут обрабатываться особым образом. В ячейках AREG, I1REG, ..., ZERO всегда будут храниться абсолютные значения величин, содержащихся в соответствующих регистрах. В другом наборе ячеек, использующемся в данной программе, (SIGNA, SIGN1, ..., SIGNZ), будут содержаться значения $+1$ или -1, в зависимости от знака соответствующего регистра (т. е. "плюс" это или "минус").

В программе-интерпретаторе, как правило, есть раздел, который представляет собой орган центрального управления. Он вступает в действие между интерпретируемыми командами. В нашем случае после выполнения каждой сымитированной команды программа переходит к ячейке CYCLE.

Управляющая программа выполняет одинаковые для всех команд действия; она разделяет команду на составные части и помещает эти части в такие места, откуда их будет удобно выбирать для дальнейшего использования. В приведенной ниже программе используются следующие установки:

rI6 — адрес ячейки, в которой сохраняется следующая команда;
rI5 — M (адрес текущей команды с учетом индексирования);
rI4 — код операции текущей команды;
rI3 — F-поле текущей команды;
INST — текущая команда.

Программа M.

```
001  * ИМИТАТОР MIX
002          ORIG 3500          Имитируемая память, начиная с 0000 и далее.
003  BEGIN   STZ  TIME(0:2)
004          STZ  OVTOG         OVTOG — имитируемый флаг переполнения.
005          STZ  COMPI         COMPI, ±1 или 0, — флаг сравнения.
006          ENT6 0             Взять первую команду из нулевой ячейки.
007  CYCLE   LDA  CLOCK         Начало выполнения управляющей программы.
008  TIME    INCA 0             Указывает на ячейку, содержащую время вы-
009          STA  CLOCK           полнения предыдущей команды (строка 033).
010          LDA  0,6           rA ← имитируемая команда.
011          STA  INST
012          INC6 1             Увеличить значение счетчика адреса.
013          LDX  INST(1:2)     Взять абсолютное значение адреса.
014          SLAX 5             Присоединить знак к адресу.
015          STA  M
016          LD2  INST(3:3)     Проверить поле индекса.
017          J2Z  1F            Это нуль?
018          DEC2 6
019          J2P  INDEXERROR    Определен недопустимый индекс?
020          LDA  SIGN6,2       Взять знак индексного регистра.
021          LDX  I6REG,2       Взять абсолютное значение индексного регистра.
022          SLAX 5             Присоединить знак.
023          ADD  M             Сложение с учетом знаков для индексирования.
024          CMPA ZERO(1:3)     Результат слишком велик?
025          JNE  ADDRERROR     Если да, имитировать ошибку.
026          STA  M             В противном случае адрес найден.
027  1H      LD3  INST(4:4)     rI3 ← F-поле.
028          LD5  M             rI5 ← M.
029          LD4  INST(5:5)     rI4 ← C-поле.
030          DEC4 63
031          J4P  OPERROR       Код операции ≥ 64?
032          LDA  OPTABLE,4(4:4) Взять из таблицы время выполнения.
033          STA  TIME(0:2)
034          LD2  OPTABLE,4(0:2) Взять адрес соответствующей программы.
035          JNOV 0,2           Перейти к оператору.
036          JMP  0,2           (Защита от переполнений.)  ∎
```

Советую читателю обратить особое внимание на строки 034–036. “Таблица-переключатель” из 64 операторов — это составная часть имитатора, позволяющая ему быстро переходить к нужной программе для выполнения текущей команды. Это важный метод экономии времени (см. упр. 1.3.2–9).

В состоящей из 64 слов таблице-переключателе OPTABLE также хранится время выполнения для различных операторов; содержимое этой таблицы определяется в следующих строках.

```
037          NOP  CYCLE(1)      Таблица кодов операций;
038          ADD  ADD(2)            ее типичные элементы имеют вид
039          SUB  SUB(2)            “ОП программа(время)”.
040          MUL  MUL(10)
```

```
041          DIV  DIV(12)
042          HLT  SPEC(1)
043          SLA  SHIFT(2)
044          MOVE MOVE(1)
045          LDA  LOAD(2)
046          LD1  LOAD,1(2)
             ...
051          LD6  LOAD,1(2)
052          LDX  LOAD(2)
053          LDAN LOADN(2)
054          LD1N LOADN,1(2)
             ...
060          LDXN LOADN(2)
061          STA  STORE(2)
             ...
069          STJ  STORE(2)
070          STZ  STORE(2)
071          JBUS JBUS(1)
072          IOC  IOC(1)
073          IN   IN(1)
074          OUT  OUT(1)
075          JRED JRED(1)
076          JMP  JUMP(1)
077          JAP  REGJUMP(1)
             ...
084          JXP  REGJUMP(1)
085          INCA ADDROP(1)
086          INC1 ADDROP,1(1)
             ...
092          INCX ADDROP(1)
093          CMPA COMPARE(2)
             ...
100  OPTABLE CMPX COMPARE(2)  ▮
```

(Элементы, соответствующие операторам LDi, LDiN и INCi, имеют дополнительную запись “,1”, чтобы сделать поле (3:3) ненулевым. Это используется ниже, в строках 289 и 290, для указания того, что после имитации данных операций необходимо проверить размер величины, содержащейся в соответствующем индексном регистре.)

В следующей части программы-имитатора просто перечисляются ячейки, которые используются для хранения содержимого имитируемых регистров.

```
101  AREG    CON  0  Абсолютное значение регистра A.
102  I1REG   CON  0  Абсолютное значение индексных регистров.
                 ...
107  I6REG   CON  0
108  XREG    CON  0  Абсолютное значение регистра X.
109  JREG    CON  0  Абсолютное значение регистра J.
110  ZERO    CON  0  Нулевая константа для “STZ”.
111  SIGNA   CON  1  Знак регистра A.
112  SIGN1   CON  1  Знаки индексных регистров.
                 ...
```

```
117 SIGN6   CON  1
118 SIGNX   CON  1  Знак регистра X.
119 SIGNJ   CON  1  Знак регистра J.
120 SIGNZ   CON  1  Знак, сохраняемый "STZ".
121 INST    CON  0  Имитируемая команда.
122 COMPI   CON  0  Флаг сравнения.
123 OVTOG   CON  0  Флаг переполнения.
124 CLOCK   CON  0  Имитируемое время выполнения. ▮
```

А теперь рассмотрим три подпрограммы, используемые имитатором. Первой идет подпрограмма MEMORY.

Последовательность вызова: `JMP MEMORY`.
Состояние при входе: rI5 = допустимый адрес памяти (в противном случае подпрограмма перейдет к MEMERROR).
Состояние при выходе: rX = знак слова в ячейке памяти rI5; rA = абсолютное значение слова в ячейке памяти rI5.

```
125 * ПОДПРОГРАММЫ
126 MEMORY  STJ  9F         Подпрограмма выборки из памяти.
127         J5N  MEMERROR
128         CMP5 =BEGIN=    Имитируемая память находится
129         JGE  MEMERROR       в ячейках с 0000 до BEGIN − 1.
130         LDX  0,5
131         ENTA 1
132         SRAX 5          rX ← знак слова.
133         LDA  0,5(1:5)   rA ← абсолютное значение слова.
134 9H      JMP  *          Выход. ▮
```

Подпрограмма FCHECK обрабатывает спецификацию частичного поля и проверяет, имеет ли оно вид 8L + R, где $L \le R \le 5$.

Последовательность вызова: `JMP FCHECK`.
Состояние при входе: rI3 = допустимая спецификация поля (в противном случае подпрограмма перейдет к FERROR).
Состояние при выходе: rA = rI1 = L, rX = R.

```
135 FCHECK  STJ  9F         Подпрограмма проверки поля.
136         ENTA 0
137         ENTX 0,3        rAX ← спецификация поля.
138         DIV  =8=        rA ← L, rX ← R.
139         CMPX =5=        R > 5?
140         JG   FERROR
141         STX  R
142         STA  L
143         LD1  L          rI1 ← L.
144         CMPA R
145 9H      JLE  *          Выйти, если L ≤ R.
146         JMP  FERROR ▮
```

Последняя подпрограмма, GETV, находит величину V (т. е. соответствующее поле ячейки M), используемую различными операторами MIX, как определено в разделе 1.3.1.

Последовательность вызова: JMP GETV.
Состояние при входе: rI5 = допустимый адрес памяти; rI3 = допустимое поле. (Если оно недопустимо, то регистрируется ошибка, как и выше.)
Состояние при выходе: rA = абсолютное значение V; rX = знак V; rI1 = L; rI2 = −R.
Второй вход: JMP GETAV, используется только в операторах сравнения для выборки поля из регистра.

```
147 GETAV   STJ  9F      Специальный вход (см. строку 300).
148         JMP  1F
149 GETV    STJ  9F      Подпрограмма для нахождения V.
150         JMP  FCHECK  Обработать поле и установить rI1 ← L.
151         JMP  MEMORY  rA ← абсолютное значение памяти, rX ← знак.
152 1H      J1Z  2F      Знак включен в поле?
153         ENTX 1       Если нет, установите положительный знак.
154         SLA  -1,1    Обнулите все байты слева
155         SRA  -1,1        от этого поля.
156 2H      LD2N R       Сдвиг вправо
157         SRA  5,2         в соответствующую позицию.
158 9H      JMP  *       Выход.
```

А теперь рассмотрим программы для каждого оператора в отдельности. Данные программы приведены здесь для полноты изложения, поэтому читателю стоит изучить лишь некоторые из них, если, конечно, у него нет важных причин для более тщательного изучения. Рекомендуется изучить программы для операторов SUB и JUMP: это наиболее типичные примеры. Обратите внимание на то, как можно аккуратно объединить программы для аналогичных операций, а также на то, как программа JUMP использует другую таблицу-переключатель для управления этим типом перехода.

```
159 * ОТДЕЛЬНЫЕ ОПЕРАТОРЫ
160 ADD     JMP  GETV          Загрузить значение V в rA и rX.
161         ENT1 0             Пусть rI1 указывает на регистр A.
162         JMP  INC           Перейти к программе "увеличения".
163 SUB     JMP  GETV          Загрузить значение V в rA и rX.
164         ENT1 0             Пусть rI1 указывает на регистр A.
165         JMP  DEC           Перейти к программе "уменьшения".
166 *
167 MUL     JMP  GETV          Загрузить значение V в rA и rX.
168         CMPX SIGNA         Знаки одинаковы?
169         ENTX 1
170         JE   *+2           Занести в rX знак результата.
171         ENNX 1
172         STX  SIGNA         Поместить его в оба имитируемых регистра.
173         STX  SIGNX
```

```
174          MUL   AREG          Перемножить операнды.
175          JMP   STOREAX       Сохранить абсолютные значения.
176  *
177  DIV     LDA   SIGNA         Установить знак остатка.
178          STA   SIGNX
179          JMP   GETV          Загрузить значение V в rA и rX.
180          CMPX  SIGNA         Знаки одинаковы?
181          ENTX  1
182          JE    *+2           Занести в rX знак результата.
183          ENNX  1
184          STX   SIGNA         Поместить его в имитируемый rA.
185          STA   TEMP
186          LDA   AREG          Разделить операнды.
187          LDX   XREG
188          DIV   TEMP
189  STOREAX STA   AREG          Сохранить абсолютные значения.
190          STX   XREG
191  OVCHECK JNOV  CYCLE         Только что произошло переполнение?
192          ENTX  1             Если да, установить имитируемый
193          STX   OVTOG             флаг переполнения в положение 1.
194          JMP   CYCLE         Вернуться в управляющую программу.
195  *
196  LOADN   JMP   GETV          Загрузить значение V в rA и rX.
197          ENT1  47,4          rI1 ← C − 16; указывает регистр.
198  LOADN1  STX   TEMP          Сделать знак отрицательным.
199          LDXN  TEMP
200          JMP   LOAD1         Заменить LOADN на LOAD.
201  LOAD    JMP   GETV          Загрузить значение V в rA и rX.
202          ENT1  55,4          rI1 ← C − 8, указывает регистр.
203  LOAD1   STA   AREG,1        Сохранить абсолютные значения.
204          STX   SIGNA,1       Сохранить знак.
205          JMP   SIZECHK       Проверить, не слишком ли велико
206  *                               абсолютное значение.
207  STORE   JMP   FCHECK        rI1 ← L.
208          JMP   MEMORY        Взять содержимое ячейки памяти.
209          J1P   1F            Знак включен в поле?
210          ENT1  1             Если да, заменить L на 1
211          LDX   SIGNA+39,4        и "сохранить" знак регистра.
212  1H      LD2N  R             rI2 ← −R.
213          SRAX  5,2           Сохранить область справа от поля.
214          LDA   AREG+39,4     Вставить регистр в поле.
215          SLAX  5,2
216          ENN2  0,1           rI2 ← −L.
217          SRAX  6,2
218          LDA   0,5           Восстановить область слева от поля.
219          SRA   6,2
220          SRAX  -1,1          Присоединить знак.
221          STX   0,5           Сохранить в памяти.
222          JMP   CYCLE         Вернуться в управляющую программу.
223  *
```

```
224 JUMP     DEC3 9              Операторы перехода.
225          J3P  FERROR         F слишком велико?
226          LDA  COMPI          rA ← флаг сравнения.
227          JMP  JTABLE,3       Перейти к соответствующей программе.
228 JMP      ST6  JREG           Установить имитируемый регистр J.
229          JMP  JSJ
230          JMP  JOV
231          JMP  JNOV
232          JMP  LS
233          JMP  EQ
234          JMP  GR
235          JMP  GE
236          JMP  NE
237 JTABLE   JMP  LE             Конец таблицы перехода.
238 JOV      LDX  OVTOG          Проверить, делать ли переход по
239          JMP  *+3                переполнению.
240 JNOV     LDX  OVTOG
241          DECX 1              Взять дополнение флага переполнения.
242          STZ  OVTOG          Отключить флаг переполнения.
243          JXNZ JMP            Перейти.
244          JMP  CYCLE          Не переходить.
245 LE       JAZ  JMP            Перейти, если в rA — нуль или отрицат. число.
246 LS       JAN  JMP            Перейти, если в rA — отрицат. число.
247          JMP  CYCLE          Не переходить.
248 NE       JAN  JMP            Перейти, если в rA — отрицат. или положит.
                                      число.
249 GR       JAP  JMP            Перейти, если в rA — положит. число.
250          JMP  CYCLE          Не переходить.
251 GE       JAP  JMP            Перейти, если в rA — положит. число или нуль.
252 EQ       JAZ  JMP            Перейти, если в rA — нуль.
253          JMP  CYCLE          Не переходить.
254 JSJ      JMP  MEMORY         Проверить допустимость адресов памяти.
255          ENT6 0,5            Имитировать переход.
256          JMP  CYCLE          Вернуться в главную управляющую программу.
257 *
258 REGJUMP LDA  AREG+23,4       Регистровые переходы.
259          JAZ  *+2            В регистре нуль?
260          LDA  SIGNA+23,4     Если нет, поместить знак в rA.
261          DEC3 5
262          J3NP JTABLE,3       Заменить командой условного перехода, если
263          JMP  FERROR             F-спецификация не слишком велика.
264 *
265 ADDROP   DEC3 3              Операторы пересылки адреса.
266          J3P  FERROR         F слишком велико?
267          ENTX 0,5
268          JXNZ *+2            Найти знак M.
269          LDX  INST
270          ENTA 1
271          SRAX 5              rX ← знак M.
272          LDA  M(1:5)         rA ← абсолютное значение M.
```

```
273          ENT1 15,4              rI1 указывает регистр.
274          JMP  1F,3              Четырехсторонний переход.
275          JMP  INC               Увеличить.
276          JMP  DEC               Уменьшить.
277          JMP  LOAD1             Занести.
278  1H      JMP  LOADN1            Занести отрицательное число.
279  DEC     STX  TEMP              Изменить знак на противоположный.
280          LDXN TEMP              Заменить DEC на INC.
281  INC     CMPX SIGNA,1           Программа сложения.
282          JE   1F                Знаки одинаковы?
283          SUB  AREG,1            Нет; вычесть абсолютные значения.
284          JANP 2F                Нужно ли изменить знак?
285          STX  SIGNA,1           Изменить знак регистра.
286          JMP  2F
287  1H      ADD  AREG,1            Сложить абсолютные значения.
288  2H      STA  AREG,1(1:5)       Сохранить абсолютное значение результата.
289  SIZECHK LD1  OPTABLE,4(3:3)    Только что был загружен
290          J1Z  OVCHECK               индексный регистр?
291          CMPA ZERO(1:3)         Если да, убедиться, что результат
292          JE   CYCLE                 помещается в двух байтах.
293          JMP  SIZEERROR
294  *
295  COMPARE JMP  GETV              Загрузить значение V в rA и rX.
296          SRAX 5                 Присоединить знак.
297          STX  V
298          LDA  XREG,4            Взять поле F соответствующего регистра.
299          LDX  SIGNX,4
300          JMP  GETAV
301          SRAX 5                 Присоединить знак.
302          CMPX V                 Сравнить (заметьте, что −0 = +0).
303          STZ  COMPI             Установить флаг сравнения
304          JE   CYCLE                 в положение 0, +1
305          ENTA 1                     либо −1.
306          JG   *+2
307          ENNA 1
308          STA  COMPI
309          JMP  CYCLE             Вернуться в управляющую программу.
310  *
311          END  BEGIN
```

В приведенном выше коде используется одно хитрое правило, сформулированное в разделе 1.3.1: команда “`ENTA -0`” загружает минус нуль в регистр A, так же как команда “`ENTA -5,1`”, когда в индексном регистре 1 содержится значение +5. В общем случае, если `M` равно нулю, команда `ENTA` загружает знак этой команды, а `ENNA` загружает противоположный знак. Когда автор писал первый черновик раздела 1.3.1, он не придал особого значения определению этого условия. Такие вопросы обычно возникают только при написании компьютерной программы, в которой используются данные правила.

Несмотря на свои размеры, приведенная выше программа является неполной в нескольких отношениях.

a) Она не распознает операций с плавающей точкой.
b) Программирование кодов операций 5–7 оставлено для упражнения.
c) Программирование операторов ввода-вывода оставлено для упражнения.
d) Не предусмотрены средства загрузки имитируемых программ (см. упр. 4).
e) Не включены программы

INDEXERROR, ADDRERROR, OPERROR, MEMERROR, FERROR, SIZEERROR.

Они предназначены для обработки ошибок, обнаруженных в имитируемой программе.
f) Не предусмотрены средства диагностики. (Хороший имитатор должен, например, быть способным распечатать содержимое регистров во время выполнения программы.)

УПРАЖНЕНИЯ

1. [*14*] Изучите все варианты применения подпрограммы FCHECK в программе-имитаторе. Можете ли вы предложить более удачный способ организации программы? (См. шаг 3 в конце раздела 1.4.1.)

2. [*20*] Напишите программу SHIFT, которой не хватает в программе, приведенной в тексте (код операции 6).

3. [*22*] Напишите программу MOVE, которой не хватает в программе, приведенной в тексте (код операции 7).

4. [*14*] Измените программу в тексте раздела, чтобы она начиналась так, как будто была нажата "кнопка GO" машины MIX (см. упр. 1.3.1–26).

▸ **5.** [*24*] Определите, сколько времени потребуется для имитирования операторов LDA и ENTA по сравнению с реальным временем, которое MIX затратит на их непосредственное выполнение.

6. [*28*] Напишите программы для операторов ввода-вывода JBUS, IOC, IN, OUT и JRED, которых не хватает в программе, приведенной в тексте раздела, разрешив использовать только устройства 16 и 18. Предполагается, что операции "читать перфокарту" и "перейти к новой странице" занимают $T = 10000u$, в то время как операция "печатать строку" занимает $T = 7500u$. [*Замечание.* Опыт показывает, что команду JBUS следует имитировать, рассматривая "JBUS *" как частный случай; иначе имитатор, похоже, остановится!]

▸ **7.** [*32*] Модифицируйте решение предыдущего упражнения таким образом, чтобы выполнение команды IN или OUT не приводило к немедленной передаче входных-выходных данных. Такая передача должна происходить только после того, как пройдет примерно половина времени, которое необходимо для имитируемых устройств. (Это позволит избежать распространенной студенческой ошибки, когда команды IN и OUT используются неправильно.)

8. [*20*] Истинно или ложно следующее утверждение: "Каждый раз при выполнении строки 010 программы-имитатора выполняется неравенство $0 \le$ rI6 $<$ BEGIN"?

***1.4.3.2. Программы трассировки.** Когда машина имитируется на самой себе (как в предыдущем разделе MIX имитировался на MIX), получается частный случай имитатора, который называется программой *трассировки* или *слежения*. Подобные программы иногда используются при отладке, так как позволяют распечатать пошаговый отчет о поведении имитируемой программы.

В предыдущем разделе программа была написана так, как будто MIX имитировался на другом компьютере. Для программ трассировки используется совершенно другой подход; обычно мы позволяем, чтобы регистры представляли сами себя, а операторы выполняли сами себя, т. е. позволяем машине самой выполнять большинство команд. Основное исключение представляет команда перехода или условного перехода, которую нельзя выполнять, не модифицируя, так как программа трассировки должна сохранять общий контроль. Каждой машине присущи уникальные, свойственные только ей особенности, и это значительно усложняет трассировку. Для машины MIX самая трудная и интересная проблема связана с регистром J.

Приведенная ниже программа трассировки запускается, когда главная программа переходит к ячейке ENTER. При этом в регистре J содержится адрес, с которого трассировка должна *начаться*, а в регистре X — адрес, где она должна *закончиться*. Это интересная программа, заслуживающая внимательного изучения.

```
01  * TRACE ROUTINE
02  ENTER  STX  TEST(0:2)     Установить адрес выхода.
03         STX  LEAVEX(0:2)
04         STA  AREG          Сохранить содержимое rA.
05         STJ  JREG          Сохранить содержимое rJ.
06         LDA  JREG(0:2)     Взять начальный адрес для трассировки.
07  CYCLE  STA  PREG(0:2)     Сохранить адрес следующей команды.
08  TEST   DECA *             Это адрес выхода?
09         JAZ  LEAVE
10  PREG   LDA  *             Взять следующую команду.
11         STA  INST          Скопировать ее.
12         SRA  2
13         STA  INST1(0:3)    Сохранить этот адрес и индексы.
14         LDA  INST(5:5)     Взять код операции, C.
15         DECA 38
16         JANN 1F            C ≥ 38 (JRED)?
17         INCA 6
18         JANZ 2F            C ≠ 32 (STJ)?
19         LDA  INST(0:4)
20         STA  *+2(0:4)      Заменить STJ на STA.
21  JREG   ENTA *             rA ← имитируемое содержимое rJ.
22         STA  *
23         JMP  INCP
24  2H     DECA 2
25         JANZ 2F            C ≠ 34 (JBUS)?
26         JMP  3F
27  1H     DECA 9             Тест для команд перехода.
28         JAP  2F            C > 47 (JXL)?
29  3H     LDA  8F(0:3)       Мы обнаружили команду перехода;
30         STA  INST(0:3)         изменить ее адрес на "JUMP".
31  2H     LDA  AREG          Восстановить регистр A.
32  *                         Во всех регистрах, кроме J, теперь правильные
33  *                             значения по отношению к внешней программе.
34  INST   NOP  *             Команда выполняется.
35         STA  AREG          Снова сохранить регистр A.
36  INCP   LDA  PREG(0:2)     Перейти к следующей команде.
```

```
37          INCA 1
38          JMP  CYCLE
39  8H      JSJ  JUMP          Константа для строк 29 и 40.
40  JUMP    LDA  8B(4:5)       Встретился переход.
41          SUB  INST(4:5)     Это была JSJ?
42          JAZ  *+4
43          LDA  PREG(0:2)     Если нет, обновите имитируемый
44          INCA 1                  регистр J.
45          STA  JREG(0:2)
46  INST1   ENTA *
47          JMP  CYCLE         Перейти к адресу перехода.
48  LEAVE   LDA  AREG          Восстановить регистр A.
49  LEAVEX  JMP  *             Остановить трассировку.
50  AREG    CON  0             Содержимое имитируемого rA.
```

По поводу программ трассировки в целом и этой в частности нужно отметить следующее.

1) Здесь представлена только самая интересная часть программы трассировки — часть, которая является управляющей во время выполнения другой программы. Чтобы трассировка принесла пользу, необходима также программа записи содержимого регистров, но мы ее не включили. Такая программа, хотя и безусловно важная, отвлекает внимание от более тонких моментов программы трассировки. Поэтому модификации, которые в этой связи необходимо ввести в программу, оставлены для упражнения (см. упр. 2).

2) Занимаемое пространство обычно имеет большее значение, чем время выполнения, т. е. программа должна быть настолько короткой, насколько это возможно. Тогда программа трассировки сможет "сосуществовать" даже с очень большими программами. А время выполнения все равно расходуется на вывод данных.

3) Мы позаботились о том, чтобы избежать изменения содержимого большинства регистров. Фактически программа использует только регистр A машины MIX. Программа трассировки не оказывает влияния ни на флаг сравнения, ни на флаг переполнения. (Чем меньше регистров используется, тем меньшее их число нужно восстанавливать.)

4) Когда происходит переход к ячейке JUMP, необязательно выполнять команду "STA AREG", так как содержимое rA не изменилось.

5) После выхода из программы трассировки регистр J не восстанавливается. В упр. 1 показано, как исправить эту ситуацию.

6) На трассируемую программу налагаются только три ограничения.

a) Она не должна сохранять что-либо в ячейках, используемых программой трассировки.

b) Она не должна использовать выходное устройство, на которое выводится информация о трассировке (например, команда JBUS дала бы неправильное указание).

c) Во время трассировки она будет работать медленнее.

УПРАЖНЕНИЯ

1. [*22*] Модифицируйте программу трассировки так, чтобы при выходе она восстанавливала регистр J. (Можете предполагать, что содержимое регистра J ненулевое.)

2. [*26*] Модифицируйте приведенную в тексте программу трассировки таким образом, чтобы перед выполнением каждого шага она записывала на магнитную ленту (устройство 0) следующую информацию.

Слово 1, поле (0:2): адрес ячейки.
Слово 1, поле (4:5): регистр J (перед выполнением).
Слово 1, поле (3:3): 2, если результат сравнения — больше, 1 — если равно, 0 — если меньше; плюс 8, если перед выполнением нет переполнения.
Слово 2: команда.
Слово 3: регистр A (перед выполнением).
Слова 4–9: регистры I1–I6 (перед выполнением).
Слово 10: регистр X (перед выполнением).

Слова 11–100 каждого блока ленты размером 100 слов должны содержать еще девять групп по 10 слов, записанных в том же формате.

3. [*10*] В предыдущем упражнении программа трассировки записывает свои выходные данные на магнитную ленту. Объясните, почему это лучше, чем непосредственно распечатать результаты.

▸ **4.** [*25*] Что произойдет, если программа трассировки будет трассировать *саму себя*? Если более конкретно, то выясните, как будет работать программа трассировки, если две команды — `ENTX LEAVEX` и `JMP *+1` — поместить непосредственно перед `ENTER`.

5. [*28*] Рассмотрим задачу, аналогичную предыдущей. Пусть две копии программы трассировки помещены в различных местах памяти и настроены так, чтобы трассировать одна другую. Что произойдет?

▸ **6.** [*40*] Напишите программу трассировки, способную трассировать себя в смысле упр. 4: она должна в замедленном режиме распечатывать собственные шаги, а *та* программа будет трассировать саму себя в еще *более замедленном* режиме, и так до бесконечности, пока будет хватать памяти.

▸ **7.** [*25*] Подумайте, как написать эффективную программу *трассировки переходов*, которая выдает намного меньше выходных данных, чем обычная программа трассировки. Вместо того чтобы отображать содержимое регистров, программа трассировки переходов просто записывает происходящие переходы. Она выдает последовательность пар (x_1, y_1), (x_2, y_2), ... Это означает, что программа перешла из ячейки x_1 в y_1, затем (после выполнения команд из ячеек y_1, $y_1 + 1$, ..., x_2) — из ячейки x_2 в y_2 и т. д. [На основании этой информации следующая программа может восстановить ход выполнения программы и установить, как часто выполнялась каждая команда.]

1.4.4. Ввод и вывод

Вероятно, самое очевидное, чем один компьютер отличается от другого, — это устройства ввода-вывода, а также компьютерные команды, управляющие этими периферийными устройствами. В рамках одной книги невозможно обсудить все проблемы и методы, связанные с данной областью, поэтому мы ограничимся изучением наиболее типичных методов ввода-вывода, применимых для большинства компьютеров. Операторы ввода-вывода машины `MIX` представляют собой нечто среднее между самыми разнообразными средствами, имеющимися в реальных компьютерах. Чтобы вы могли представить себе операции ввода-вывода на конкретном примере, давайте в этом разделе обсудим проблемы оптимального выполнения операций ввода-вывода в машине `MIX`.

И снова я прошу читателя снисходительно отнестись к тому, что здесь рассматривается анахроничная машина MIX *с ее перфокартами и т. п. Хотя эти устаревшие устройства сегодня полностью вышли из употребления, они по-прежнему могут дать несколько важных уроков. Но, конечно, когда для этой цели будет использоваться компьютер* MMIX*, он преподаст нам их еще лучше.*

Многие пользователи компьютеров полагают, что ввод и вывод на самом деле не являются частью процесса "настоящего" программирования. Считается, что ввод и вывод — это нудные задачи, которые приходится выполнять только для того, чтобы ввести информацию в компьютер и вывести из него полученные результаты. Поэтому средства ввода-вывода компьютера обычно начинают изучать только после знакомства с остальными его возможностями. И часто случается так, что лишь небольшая группа программистов некоторого компьютера вообще что-либо знает о деталях операций ввода-вывода. Такое положение вещей в чем-то даже естественно, поскольку программирование ввода-вывода никогда не было особенно привлекательным. Но до тех пор, пока многие не начнут серьезно относиться к данному предмету, нельзя ожидать, что ситуация изменится. В этом и других разделах (например, в разделе 5.4.6) вы увидите, что в связи с вводом-выводом возникает множество интересных вопросов, а также узнаете о существовании некоторых изящных алгоритмов.

Теперь, пожалуй, нужно сделать небольшое отступление по поводу терминологии. В словарях английского языка слова "input" ("ввод") и "output" ("вывод") раньше приводились только как существительные ("What kind of input are we getting?" — "Какого вида данные мы вводим?"). Но теперь уже вошло в привычку использовать их как прилагательные ("Don't drop the input tape" — "Не сотрите эту ленту с входными данными") и переходные глаголы ("Why did the program output this garbage?" — "Почему программа выводит эту чепуху?"). Вместо комбинированного термина "ввод-вывод" чаще всего используют аббревиатуру "В/В" (на английском — "I/O"). Операцию ввода часто называют *чтением*, а вывода соответственно *записью*. Элементы, составляющие ввод или вывод, обычно называют "данными" (на английском — "data"). В английском языке это слово, строго говоря, является формой множественного числа (как и в русском языке. — *Прим. перев.*), но используется в собирательном смысле, как будто это единственное число ("The data has not been read."). Точно так и слово "information" ("информация") является формой как единственного, так и множественного числа. На этом урок английского закончен.

Предположим, необходимо прочитать данные с магнитной ленты. Оператор `IN` машины `MIX`, как описано в разделе 1.3.1, просто *инициирует* процесс ввода, и в то время, пока данные вводятся, компьютер продолжает выполнять следующие команды. Поэтому по команде "`IN 1000(5)`" начинается считывание 100 слов с накопителя на магнитных лентах под номером 5 в ячейки памяти 1000–1099, но последующие команды программы пока не должны обращаться к этим ячейкам. Программа может считать, что ввод завершен только после того, как (a) инициируется другая операция В/В (`IN`, `OUT` или `IOC`), обращающаяся к устройству 5, или (b) команда условного перехода `JBUS(5)` или `JRED(5)` показывает, что устройство 5 больше не "занято".

Поэтому самый простой способ считать блок данных с ленты в ячейки 1000–1099 так, чтобы информация была на месте, — воспользоваться двумя командами:

```
IN 1000(5); JBUS *(5).        (1)
```

Этот элементарный метод применялся в программе из раздела 1.4.2 (см. строки 07–08 и 52–53). Но он слишком неэкономно расходует время работы компьютера, так как большая часть времени, которую можно было бы использовать для вычислений, скажем $1000u$ или даже $10000u$, тратится на многократное повторение команды "JBUS". Если это время использовать для вычислений, то скорость выполнения программы можно удвоить. (См. упр. 4 и 5.)

Один из способов избежать такого "цикла ожидания" — использовать две области памяти для ввода. Можно считывать данные в одну область, в то же время выполняя вычисления над данными в другой. Например, программу можно начать командой

```
IN    2000(5)    Начать чтение первого блока.                    (2)
```

И теперь каждый раз, когда понадобится прочитать блок с ленты, можно дать пять следующих команд.

```
ENT1  1000       Подготовиться к выполнению оператора MOVE.
JBUS  *(5)       Ожидать готовности устройства 5.
MOVE  2000(50)   (2000–2049) → (1000–1049).                      (3)
MOVE  2050(50)   (2050–2099) → (1050–1099).
IN    2000(5)    Начать чтение следующего блока.
```

В целом, это дает такой же эффект, как и применение команды (1), но лента с входными данными остается занятой, пока программа работает над данными, находящимися в ячейках 1000–1099.

Последняя команда (3) начинает считывание блока данных с ленты в ячейки 2000–2099 до того, как был обработан предыдущий блок. Это называется досрочным чтением или *опережающим вводом* и делается в расчете на то, что данный блок в конце концов понадобится. Но на самом деле, начав обработку блока в ячейках 1000–1099, можно обнаружить, что никаких данных больше не нужно. Мы уже встречались с аналогичной ситуацией, например, в сопрограмме из раздела 1.4.2, где входные данные поступали с перфокарт, а не с ленты. Появление "." в любом месте перфокарты означало, что это последняя карта колоды. Подобная ситуация делает опережающий ввод невозможным. Исключение составляют случаи, когда можно предположить, что (a) за колодой перфокарт с входными данными следует пустая перфокарта или специальная дополнительная карта некоторого другого вида, либо (b), скажем, в колонке 80 последней карты колоды появляется опознавательная метка (например, "."). При использовании опережающего ввода для правильного завершения ввода в конце программы всегда должны быть предусмотрены специальные средства.

Метод параллельного выполнения вычислений и операций В/В называется *буферизацией*, в то время как элементарный метод (1) называется *небуферизированным* вводом. Область ячеек памяти 2000–2099, которая используется для сохранения опережающего ввода в (3), а также область ячеек 1000–1099, в которые перемещаются входные данные, называется *буфером*. В словаре Вебстера (Webster) *New World Dictionary* слово "буфер" определяется как "Любой человек или предмет, который служит для смягчения удара". Этот термин нам подходит, так как для буферизации характерна более ровная работа устройств В/В. (Инженеры-электронщики часто используют слово "буфер" в другом смысле, обозначая им часть устройства В/В, в которой сохраняется информация во время ее передачи. Но в этой книге термин

"буфер" будет означать область *памяти*, используемую программистом для хранения данных В/В.)

Последовательность (3) не всегда лучше, чем (1), хотя исключения из этого правила достаточно редки. Давайте сравним их время выполнения. Пусть T — время, необходимое для ввода 100 слов, и пусть C — время выполнения, которое проходит между запросами на ввод данных. Для метода (1) требуется, в основном, $T + C$ времени на блок ленты, а для метода (3) — в основном, $\max(C, T) + 202u$. (Величина $202u$ — это время, необходимое для выполнения двух команд `MOVE`.) Один из способов оценки времени выполнения состоит в том, чтобы рассмотреть время прохождения критического пути; в данном случае — промежуток времени между моментами использования устройства В/В, в течение которого оно не занято. В методе (1) устройство остается незанятым в течение C единиц времени, а в методе (3) — 202 единиц времени (в предположении, что $C < T$).

Относительно медленно работающие команды `MOVE` из (3) использовать нежелательно, особенно потому, что они отнимают время прохождения критического пути, когда накопитель на магнитной ленте должен быть неактивным. Но существует почти очевидный способ улучшения этого метода, который позволит избежать использования команд `MOVE`. Внешнюю программу можно переделать так, чтобы она обращалась поочередно к ячейкам 1000–1099 и 2000–2099. Во время считывания данных в один буфер можно выполнять вычисления в другом буфере; затем можно начать считывание в другой буфер, в то же время проводя вычисления над данными в первом буфере. Этот важный метод называется *свопингом*. Адрес текущего буфера сохраняется в индексном регистре (или, если нет свободных индексных регистров, в ячейке памяти). Мы уже встречались с методом свопинга, когда он применялся к выходным данным алгоритма 1.3.2Р (см. шаги Р9–Р11) и к сопутствующей программе.

Рассмотрим пример применения метода свопинга ко вводу. Предположим, каждый блок ленты состоит из 100 отдельных элементов, содержащих по одному слову. Ниже приведена подпрограмма, которая берет следующее слово из входных данных и начинает считывание нового блока, если текущий исчерпан.

```
01  WORDIN STJ  1F          Сохранить адрес выхода.
02         INC6 1           Перейти к следующему слову.
03  2H     LDA  0,6         Это конец
04         CMPA =SENTINEL=      буфера?
05  1H     JNE  *           Если нет, выйти.
06         IN   -100,6(U)   Пополнить этот буфер.
07         LD6  1,6         Переключиться на другой
08         JMP  2B              буфер и вернуться.
09  INBUF1 ORIG *+100       Первый буфер.
10         CON  SENTINEL    Маркер конца буфера.
11         CON  *+1         Адрес другого буфера.
12  INBUF2 ORIG *+100       Второй буфер.
13         CON  SENTINEL    Маркер конца буфера.
14         CON  INBUF1      Адрес другого буфера.  ∎
```
(4)

В этой программе для адресации последнего слова ввода используется регистр 6. Предполагается, что вызывающая программа не изменяет его значение. Символ `U`

обозначает накопитель на магнитной ленте, а символ SENTINEL — это значение, о котором известно (из характеристик программы), что его *нет* ни в одном блоке на ленте.

По поводу данной подпрограммы необходимо отметить следующее.

1) Константа-маркер (sentinel)появляется в качестве 101-го слова каждого буфера и представляет собой удобное средство для обозначения окончания буфера. Но во многих задачах этот метод не будет надежным, так как на ленте может появиться любое слово. Если вводить данные с перфокарт, то подобный метод (когда 17-е слово буфера является маркером) всегда можно использовать, не опасаясь сбоя. В этом случае маркером может служить любое отрицательное число, так как при вводе в MIX с перфокарт всегда получаются неотрицательные слова.

2) В каждом буфере содержится адрес другого буфера (см. строки 07, 11 и 14). Эта "взаимосвязь" облегчает процесс свопинга.

3) Нет необходимости в команде JBUS, так как следующий ввод инициируется до того, как происходит обращение к какому-либо слову из предыдущего блока. Если величины C и T, как и раньше, обозначают время вычисления и время ввода данных с ленты, то время выполнения из расчета на блок ленты теперь составляет $\max(C,T)$. Поэтому, если $C \leq T$, ленту можно оставить работать на полной скорости. (*Примечание.* В данном отношении MIX — идеализированный компьютер, так как программа не должна обрабатывать никаких ошибок В/В. На большинстве машин непосредственно перед командой IN в этой программе необходимо было бы использовать команды проверки успешного завершения предыдущей операции.)

4) Чтобы подпрограмма (4) работала правильно, необходимо запустить еще кое-что, как только программа начнет работать. Мы предоставляем читателю самостоятельно разобраться в деталях (см. упр. 6).

5) Благодаря подпрограмме WORDIN для всей остальной программы дело выглядит таким образом, как будто блоки на магнитной ленте имеют длину, равную 1, а не 100. Такой способ разбиения одного блока на ленте на несколько программно-ориентированных записей называется *блокировкой записей*.

Методы, продемонстрированные здесь для ввода, с небольшими изменениями применимы и для вывода (см. упр. 2 и 3).

Множественная буферизация. Свопинг — это только частный случай при $N = 2$ общего метода для N буферов. В некоторых задачах желательно иметь более двух буферов. Для примера рассмотрим следующий тип алгоритма.

Шаг 1. Прочитать пять блоков один за другим.

Шаг 2. На основе этих данных выполнить достаточно трудоемкие вычисления.

Шаг 3. Вернуться к шагу 1.

В данном случае желательно иметь пять-шесть буферов, чтобы во время выполнения шага 2 можно было читать следующую порцию данных из пяти блоков. Благодаря этой тенденции передавать для В/В пакеты данных множественная буферизация получает большие преимущества по сравнению со свопингом.

Предположим, имеется N буферов для некоторого процесса ввода или вывода, выполняющегося с помощью единственного устройства В/В. Будем представлять

себе, что эти буфера расположены по кругу, как показано на рис. 23. Можно считать, что внешняя для процесса буферизации программа имеет следующий общий вид относительно нашего устройства В/В.

```
⋮
НАЗНАЧИТЬ
⋮
ОСВОБОДИТЬ
⋮
НАЗНАЧИТЬ
⋮
ОСВОБОДИТЬ
⋮
```

Другими словами, можно считать, что в программе чередуются действие под названием НАЗНАЧИТЬ и действие под названием ОСВОБОДИТЬ, между которыми выполняются другие вычисления, не оказывающие влияния на распределение буферов.

НАЗНАЧИТЬ означает, что программа получает адрес следующей буферной области; этот адрес присваивается как значение некоторой переменной, использующейся в программе.

ОСВОБОДИТЬ означает, что программа закончила работу с текущей буферной областью.

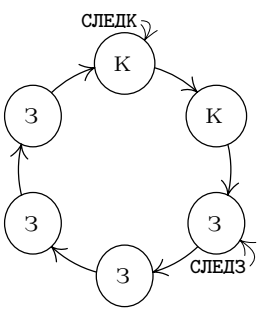


Рис. 23. Кольцо буферов ($N = 6$).

Между НАЗНАЧИТЬ и ОСВОБОДИТЬ программа работает с одним из буферов, который называется *текущей* буферной областью, а между ОСВОБОДИТЬ и НАЗНАЧИТЬ программа не обращается ни к одной буферной области.

Понятно, что НАЗНАЧИТЬ может непосредственно следовать за ОСВОБОДИТЬ. Рассмотрение вопросов буферизации часто основывалось на этом предположении. Но если ОСВОБОДИТЬ выполняется как можно быстрее, то процесс буферизации является более независимым и более эффективным. Разделяя эти две, по существу, разные функции, т. е. НАЗНАЧИТЬ и ОСВОБОДИТЬ, мы обнаружим, что метод буферизации остается легким для восприятия, а наше обсуждение — содержательным даже при $N = 1$.

Для большей определенности давайте рассмотрим операции ввода и вывода по отдельности. Для операции ввода будем предполагать, что мы имеем дело с устрой-

ством чтения перфокарт. Действие НАЗНАЧИТЬ означает, что программе нужна информация с новой перфокарты. Поэтому следует занести в индексный регистр адрес ячейки памяти, в которой находится образ следующей карты. Действие ОСВОБОДИТЬ выполняется, когда информация об образе текущей перфокарты больше не нужна, так как она некоторым способом обработана программой (возможно, скопирована в другую часть памяти и т. д.). Поэтому текущую буферную область теперь можно заполнить следующей порцией опережающего ввода.

Для операции вывода рассмотрим АЦПУ. Действие НАЗНАЧИТЬ происходит, когда нужна свободная буферная область, в которую помещается образ строки для печати. Мы хотим занести в индексный регистр адрес ячейки памяти, с которой начинается такая область. Действие ОСВОБОДИТЬ происходит, когда этот образ строки полностью помещен в буферную область в виде, готовом для печати.

Пример. Чтобы напечатать содержимое ячеек 0800–0823, можно записать следующее.

```
JMP  ASSIGNP    (Занести в rI5 адрес буфера.)
ENT1 0,5
MOVE 800(24)    Переместить 24 слова в выходной буфер.
JMP  RELEASEP
```
(5)

Здесь ASSIGNP и RELEASEP — это подпрограммы, выполняющие две описанные функции буферизации для АЦПУ.

В оптимальной ситуации с точки зрения компьютера для выполнения операции НАЗНАЧИТЬ времени практически не требуется. Для ввода это означает, что каждый образ перфокарты будет введен с опережением, так что данные уже имеются в наличии, когда программа готова их принять. А для вывода это означает, что в памяти всегда будет свободное место для записи образа строки. В любом случае время на ожидание устройств В/В не тратится.

Чтобы получше описать алгоритм буферизации и сделать его более красочным, будем говорить, что буферная область либо зеленая, либо желтая, либо красная (на рис. 24 они обозначены буквами З, Ж и К).

Зеленый цвет означает, что область готова для того, чтобы ее НАЗНАЧИЛИ, т. е. заполнена информацией с опережением (в случае ввода) либо является свободной (в случае вывода).

Желтый цвет означает, что область уже НАЗНАЧЕНА, но еще не СВОБОДНА, т. е. это текущий буфер, с которым работает программа.

Красный цвет означает, что область уже СВОБОДНА, т. е. это свободная область (в случае ввода) или область, заполненная информацией (в случае вывода).

На рис. 23 изображены два “указателя”, направленных на обозначенные кружочками буферные области. Это, по сути, индексные регистры в программе. СЛЕДЗ и СЛЕДК расшифровываются как “следующий зеленый” и “следующий красный” буфер соответственно. Третий указатель, ТЕКУЩИЙ (рис. 24), указывает на желтый буфер, если он есть.

Приведенные ниже алгоритмы можно с одинаковым успехом применять как для ввода, так и для вывода, но для определенности сначала рассмотрим случай ввода с устройства чтения перфокарт. Предположим, что программа достигла состояния,

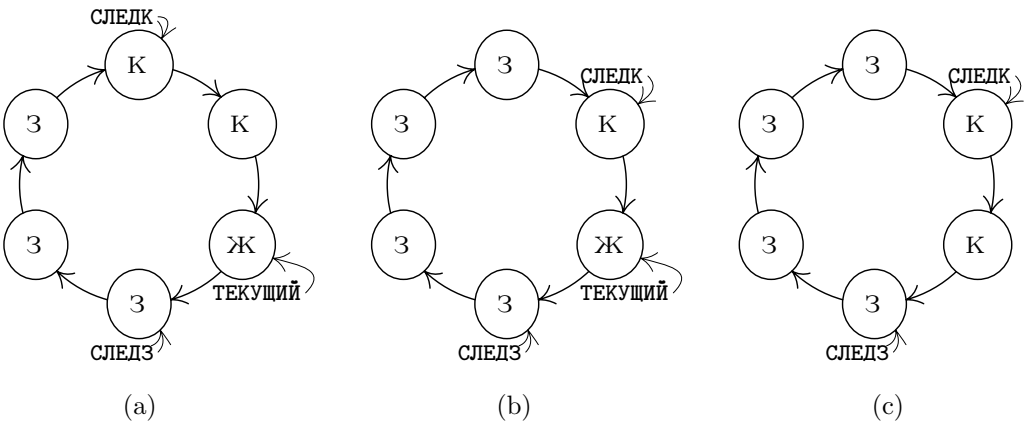


Рис. 24. Состояния буферов: (a) после НАЗНАЧИТЬ, (b) после завершения В/В и (c) после ОСВОБОДИТЬ.

показанного на рис. 23. Это означает, что четыре образа перфокарт были введены с опережением с помощью процесса буферизации и находятся в зеленых буферах. В этот момент *одновременно* происходят два действия: (a) программа проводит вычисления по командам, следующим за ОСВОБОДИТЬ, и (b) перфокарта считывается в буфер, на который указывает СЛЕДК. Это состояние дел сохранится до тех пор, пока не завершится цикл ввода (и тогда устройство перейдет из состояния "занято" в состояние "готово") или пока программа не выполнит операцию НАЗНАЧИТЬ. Предположим, что сначала произошло последнее. Тогда буфер, на который указывает стрелка СЛЕДЗ, станет желтым (т. е. текущим), стрелка СЛЕДЗ сместится по часовой стрелке и получится конфигурация, изображенная на рис. 24, (a). Если к этому моменту ввод завершится, то останется еще один блок, введенный с опережением. Поэтому красный буфер станет зеленым, и стрелка СЛЕДК переместится, как показано на рис. 24, (b). Если следующей идет операция ОСВОБОДИТЬ, то получится конфигурация, изображенная на рис. 24, (c).

Пример, касающийся вывода, приведен на рис. 27 на с. 264. Здесь "цвета" буферных областей представлены как функция от времени. При этом рассматривается программа, которая начинает работу с четырех быстрых выводов, затем медленно выдает еще четыре вывода и в конце концов выдает два вывода подряд. В этом примере используются три буфера.

Указатели СЛЕДК и СЛЕДЗ весело продвигаются по кругу по часовой стрелке, каждый со своей скоростью. Это состязание между программой (которая делает зеленые буфера красными) и процессами буферизации В/В (которые делают красные буфера зелеными). В подобном случае могут возникнуть две конфликтные ситуации.

a) Если СЛЕДЗ пытается обогнать СЛЕДК, то программа опережает устройство В/В и вынуждена ждать, пока оно будет готово.
b) Если СЛЕДК пытается обогнать СЛЕДЗ, то устройство В/В опережает программу и необходимо остановить его до тех пор, пока не будет выполнена следующая операция RELEASE.

Обе эти ситуации отображены на рис. 27 (см. упр. 9).

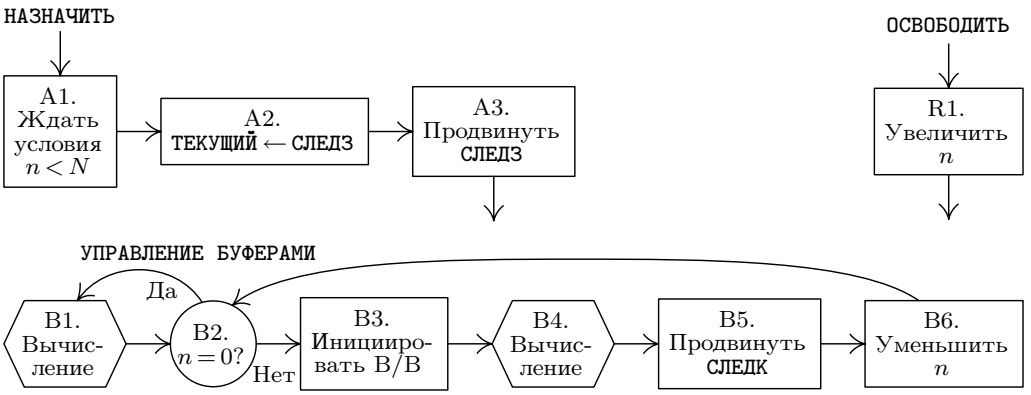


Рис. 25. Алгоритмы множественной буферизации.

К счастью, несмотря на довольно длинное объяснение, которое только что было приведено по поводу кольца буферов, сами алгоритмы обработки данной ситуации довольно просты. В следующем описании будем использовать такие обозначения:

$$\begin{aligned} N &\text{ — общее число буферов;} \\ n &\text{ — текущее число красных буферов.} \end{aligned} \tag{6}$$

В приведенном ниже алгоритме переменная n используется для того, чтобы СЛЕДЗ и СЛЕДК не мешали один другому.

Алгоритм A (НАЗНАЧИТЬ). Этот алгоритм (рис. 25) состоит из шагов, которые определяются операцией НАЗНАЧИТЬ в вычислительной программе, как описано выше.

A1. [Ждать условия $n < N$.] Если $n = N$, остановить программу, пока не будет выполнено условие $n < N$. (Если $n = N$, следовательно, ни один буфер не готов к тому, чтобы быть назначенным. Но приведенный ниже алгоритм B, работающий параллельно с данным алгоритмом в конце концов достигнет успеха в получении зеленого буфера.)

A2. [ТЕКУЩИЙ ← СЛЕДЗ.] Присвоить CURRENT ← NEXTG (назначая, таким образом, текущий буфер).

A3. [Продвинуть СЛЕДЗ.] Продвинуть СЛЕДЗ к следующему буферу по часовой стрелке. ▮

Алгоритм R (ОСВОБОДИТЬ). Этот алгоритм состоит из шагов, которые определяются операцией ОСВОБОДИТЬ в вычислительной программе, как описано выше.

R1. [Увеличить n.] Увеличить n на единицу. ▮

Алгоритм B (*Управление буферами*). Этот алгоритм осуществляет реальную инициацию операторов В/В в машине; он должен выполняться одновременно с главной программой, в том смысле, как описано ниже.

B1. [Вычисление.] Позволить главной программе в течение короткого времени проводить вычисления. Шаг B2 выполняется после некоторой задержки, когда устройство В/В готово для следующей операции.

B2. [$n = 0$?] Если $n = 0$, перейти к B1. (Таким образом, если нет ни одного красного буфера, нельзя выполнить ни одну операцию В/В.)

B3. [Инициировать В/В.] Инициировать передачу данных между буферной областью, на которую указывает СЛЕДК, и устройством В/В.

B4. [Вычисление.] Позволить главной программе работать в течение некоторого времени; затем, когда операция В/В будет завершена, перейти к шагу B5.

B5. [Продвинуть СЛЕДК.] Продвинуть СЛЕДК к следующему буферу по часовой стрелке.

B6. [Уменьшить n.] Уменьшить n на единицу и перейти к шагу B2. ▮

В этих алгоритмах присутствуют два независимых процесса, выполняющихся "одновременно", — программа управления буферизацией и вычислительная программа. Фактически эти процессы являются *сопрограммами*, которые мы назовем CONTROL и COMPUTE*. Сопрограмма CONTROL вызывает COMPUTE на шагах B1 и B4; сопрограмма COMPUTE вызывает CONTROL с помощью команд "перехода при готовности", которые разбросаны в программе через случайные промежутки.

Запрограммировать этот алгоритм для MIX чрезвычайно просто. Для удобства будем считать, что буфера связаны таким образом, что слово, *предшествующее* каждому из них, — это адрес следующего буфера. Например, при $N = 3$ имеем CONTENTS(BUF1 − 1) = BUF2, CONTENTS(BUF2 − 1) = BUF3 и CONTENTS(BUF3 − 1) = BUF1.

Программа A (НАЗНАЧИТЬ*; подпрограмма в сопрограмме* COMPUTE). rI4 ≡ ТЕКУЩИЙ; rI6 ≡ n; последовательность вызова: JMP ASSIGN; на выходе в rX содержится СЛЕДЗ.

```
НАЗНАЧИТЬ STJ  9F
1H        JRED CONTROL(U)  A1. Ожидать условия n < N.
          CMP6 =N=
          JE   1B
          LD4  СЛЕДЗ       A2. ТЕКУЩИЙ ← СЛЕДЗ.
          LDX  -1,4        A3. Продвинуть СЛЕДЗ.
          STX  СЛЕДЗ
9H        JMP  *           Выход. ▮
```

Программа R (ОСВОБОДИТЬ*; используется в сопрограмме* COMPUTE). rI6 ≡ n. Этот маленький фрагмент следует вставлять везде, где нужно выполнить операцию RELEASE.

```
          INC6 1           R1. Увеличить n.
          JRED CONTROL(U)  Возможен переход к сопрограмме CONTROL. ▮
```

Программа B (*Сопрограмма* CONTROL). rI6 ≡ n, rI5 ≡ СЛЕДК.

```
CONT1     JMP  COMPUTE     B1. Вычисление.
1H        J6Z  *-1         B2. n = 0?
          IN   0,5(U)      B3. Инициация В/В.
          JMP  COMPUTE     B4. Вычисление.
          LD5  -1,5        B5. Продвинуть СЛЕДК.
          DEC6 1           B6. Уменьшить n.
          JMP  1B ▮
```

* "Управление" и "вычисление". — *Прим. перев.*

Помимо приведенного выше кода, имеем также обычные команды связи сопрограмм:

```
CONTROL   STJ  COMPUTEX      COMPUTE   STJ  CONTROLX
CONTROLX  JMP  CONT1         COMPUTEX  JMP  COMP1    .
```

Кроме того, команду "`JRED CONTROL(U)`" следует помещать внутри программы `COMPUTE` примерно через каждые 50 команд.

Таким образом, программы множественной буферизации состоят, в сущности, только из семи команд для `CONTROL`, восьми — для `НАЗНАЧИТЬ` и двух — для `ОСВОБОДИТЬ`.

Здесь есть один замечательный факт: *один и тот же* алгоритм можно применять и для ввода, и для вывода. Но в чем же отличие между этими двумя процессами? Как программа управления определяет, что нужно делать — опережать (в случае ввода) или запаздывать (в случае вывода)? Ответ на этот вопрос кроется в начальных условиях. При вводе мы начинаем с $n = N$ (все буфера красные), а при выводе — с $n = 0$ (все буфера зеленые). И после того как программа запустится при соответствующих начальных условиях, она уже будет себя вести либо как процесс ввода, либо как процесс вывода. Другим начальным условием является `NEXTR` = `NEXTG`, т. е. обе стрелки должны указывать на один из буферов.

Для завершения программы необходимо остановить процесс ввода-вывода (если это ввод) или подождать, пока он закончится (если это вывод); подробности мы оставляем читателю (см. упр. 12 и 13).

Здесь возникает важный вопрос: "Какое значение N является оптимальным?". Конечно, по мере увеличения N скорость работы программы не уменьшится, но и не будет неограниченно расти, поэтому рано или поздно скорость роста снизится. Давайте снова обратимся к величинам C и T, которые представляют время вычислений между операторами В/В и само время В/В. Точнее, пусть C — это промежуток времени между последовательными операциями `НАЗНАЧИТЬ`, а T — промежуток времени, за которое передается один блок. Если C всегда *больше* T, то вполне достаточно будет значения $N = 2$, так как нетрудно показать, что с двумя буферами компьютер будет постоянно занят. Если C всегда *меньше* T, то и в такой ситуации достаточно значения $N = 2$, поскольку устройство В/В будет постоянно занято (за исключением случая, когда устройство имеет особые ограничения на время использования, как в упр. 19). Следовательно, бóльшие значения N полезны, главным образом, тогда, когда C принимает то малые, то большие значения. При этом, если большие значения C существенно превосходят T, подходящим значением N является среднее число последовательных малых величин C плюс 1. (Нужно заметить, однако, что преимущества буферизации фактически сводятся на нет, если ввод полностью выполняется в начале программы, а вывод — полностью в конце.) Если промежуток времени между операциями `НАЗНАЧИТЬ` и `ОСВОБОДИТЬ` всегда достаточно мал, то во всех описанных выше случаях значение N можно уменьшить на 1, что почти не окажет влияния на время выполнения.

Этот подход к буферизации можно изменять различными способами, и мы кратко расскажем о некоторых из них. До сих пор предполагалось, что используется только одно устройство В/В, но, конечно, на практике вы будете применять несколько устройств одновременно.

Существует несколько подходов к ситуации, когда используется несколько устройств. В простейшем случае у нас будет отдельное кольцо буферов для каждого устройства. Для каждого устройства будут заданы свои значения n, N, СЛЕДК, СЛЕДЗ и ТЕКУЩИЙ, а также собственная сопрограмма CONTROL. В результате мы будем иметь эффективную буферизацию одновременно на всех устройствах В/В.

Можно также создать "пул" буферных областей одинакового размера, чтобы два или более устройств совместно использовали буфера из общего списка. Это можно осуществить с помощью методов связывания памяти, которые описываются в главе 2, когда все красные буфера ввода связываются в один список, а все зеленые буфера вывода — в другой. В данном случае возникает необходимость отличать ввод от вывода, т. е. нужно переписать алгоритмы, чтобы в них не использовались n и N. Если все буфера в пуле окажутся заполненными входными данными, полученными методом опережающего ввода, то алгоритм будет неизменно останавливаться. Поэтому необходимо убедиться, что существует по крайней мере один буфер (причем желательно по одному для каждого устройства), который не является зеленым буфером ввода. И только если программа COMPUTE останавливается на шаге А1 для некоторого устройства ввода, следует разрешить ввод данных с этого устройства в последний буфер пула.

Некоторые машины имеют дополнительные ограничения на использование устройств ввода-вывода, делающие невозможной одновременную передачу данных с определенных пар устройств. (Например, несколько устройств можно подключить к компьютеру с помощью единственного "канала".) Это ограничение также оказывает влияние на нашу программу буферизации. Как сделать правильный выбор, когда нужно решить, какое устройство В/В инициировать следующим? Это называется задачей прогнозирования. Лучшее правило прогнозирования для общего случая, видимо, состоит в том, чтобы отдать предпочтение устройству, буферное кольцо которого имеет наибольшее значение n/N. При этом предполагается, что число буферов в кольце было выбрано обдуманно.

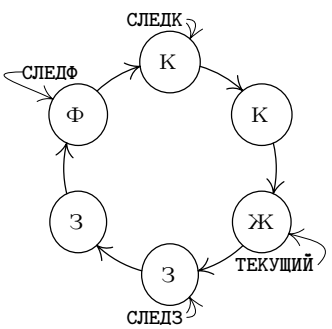


Рис. 26. Ввод и вывод в одном и том же кольце буферов.

И в заключение рассмотрим полезный метод выполнения и ввода, и вывода с помощью *одного и того же* кольца буферов, который можно применять при определенных условиях. На рис. 26 показан новый вид буфера — фиолетовый. В этой ситуации зеленые буфера представляют опережающий *ввод*; в результате операции НАЗНАЧИТЬ зеленый буфер становится желтым, а затем, после операции ОСВОБОДИТЬ, — красным и представляет блок, который нужно *вывести*. Процессы ввода и вывода, как и раньше, независимо двигаются по кругу, только сейчас после выполнения операции

вывода красные буфера становятся фиолетовыми, а при вводе фиолетовый буфер превращается в зеленый. Необходимо гарантировать, что указатели СЛЕДЗ, СЛЕДК и СЛЕДФ не будут обгонять друг друга. В момент, показанный на рис. 26, программа выполняет вычисления между операциями НАЗНАЧИТЬ и ОСВОБОДИТЬ, работая с желтым буфером; одновременно производится ввод в буфер, на который указывает стрелка СЛЕДФ, и вывод из буфера, на который указывает стрелка СЛЕДК.

УПРАЖНЕНИЯ

1. [*05*] Будет ли последовательность команд (3) по-прежнему правильной, если поместить команды MOVE перед командой JBUS, а не после нее? А если поместить команды MOVE после команды IN?

2. [*10*] С помощью команд "OUT 1000(6); JBUS *(6)" можно вывести блок данных на ленту, не используя буферизацию, точно так же, как команды (1) делают это в случае ввода. Предложите метод, аналогичный (2) и (3), который буферизирует этот вывод, используя команды MOVE и вспомогательный буфер, занимающий ячейки 2000–2099.

▶ **3.** [*22*] Напишите подпрограмму вывода с помощью свопинга, аналогичную (4). Эта подпрограмма с именем WORDOUT должна сохранять слово в rA в качестве следующего слова вывода, а когда буфер будет заполнен, записывать 100 слов на магнитную ленту (устройство V). В индексном регистре 5 должен храниться адрес текущего буфера. Составьте схему расположения буферных областей и объясните, какие команды необходимо (если необходимо вообще) использовать в начале и в конце программы, чтобы первый и последний блоки наверняка были записаны правильно. В случае необходимости последний блок следует заполнить нулями.

4. [*M20*] Покажите, что если программа использует единственное устройство В/В, то при благоприятных обстоятельствах можно сократить время ее выполнения наполовину путем буферизации В/В. Но нельзя более чем в два раза сократить время выполнения по сравнению с небуферизированным В/В.

▶ **5.** [*M21*] Обобщите решение предыдущего упражнения для случая, когда программа работает не с одним, а с n устройствами В/В.

6. [*12*] Какие команды нужно поместить в начале программы, чтобы подпрограмма WORDIN (4) начала правильно работать? (Например, в индексном регистре 6 должно *что-то* содержаться.)

7. [*22*] Напишите подпрограмму с именем WORDIN, которая, в основном, аналогична (4), за исключением того, что в ней не используется маркер конца блока.

8. [*11*] В тексте раздела описывается гипотетический сценарий ввода, начало которого показано на рис. 23, а продолжение — на рис. 24, (a), (b) и (c). Дайте интерпретацию такому же сценарию, но при условии, что выполняется вывод на АЦПУ, а не ввод с перфокарт. (Например, что происходит в момент, показанный на рис. 23?)

▶ **9.** [*21*] Программу, в результате выполнения которой содержимое буферов выглядит, как показано на рис. 27, можно охарактеризовать с помощью следующего списка промежутков времени:

$$A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000,$$
$$A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000,$$
$$A, 1000, R, 1000, A, 2000, R, 1000.$$

Этот список расшифровывается так: "Назначить, вычислять в течение $1000u$, освободить, вычислять в течение $1000u$, назначить, ..., вычислять в течение $2000u$, освободить, вычислять в течение $1000u$". Приведенные промежутки времени вычислений не включают

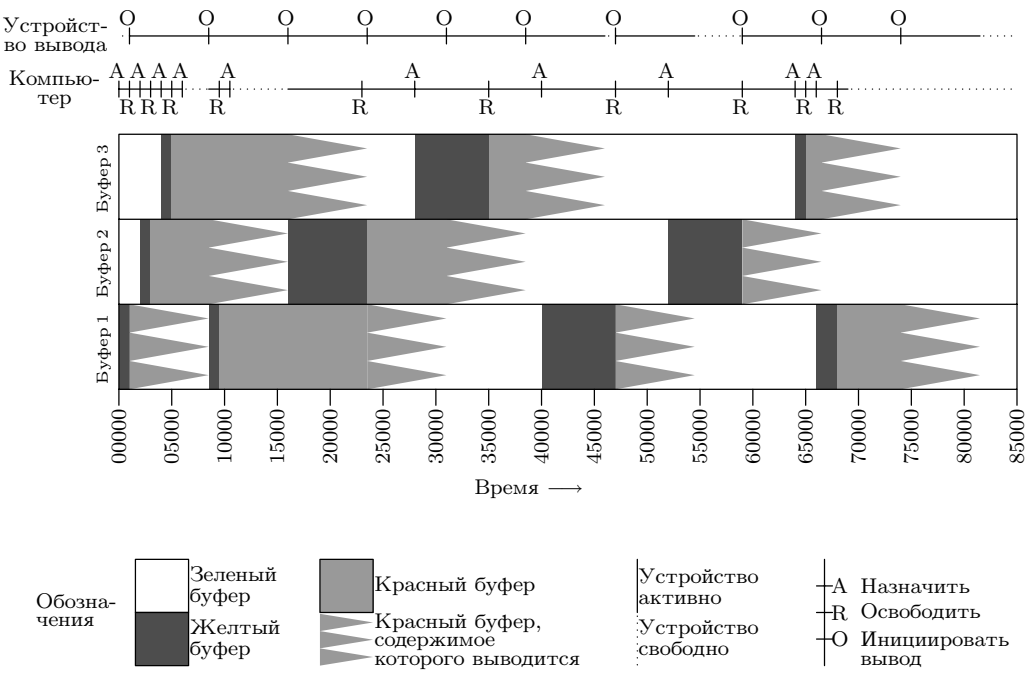


Рис. 27. Вывод с использованием трех буферов (см. упр. 9).

какие-либо периоды, когда компьютер ожидает устройство вывода, чтобы занять его (как в случае четвертой операции “назначить” на рис. 27). Выходное устройство затрачивает на вывод каждого блока по $7500u$ времени.

В следующей таблице перечислены действия, которые соответствуют промежуткам времени, показанным на рис. 27.

Время	Действие	Время	Действие
0	`ASSIGN(BUF1)`	38500	`OUT BUF3`
1000	`RELEASE, OUT BUF1`	40000	`ASSIGN(BUF1)`
2000	`ASSIGN(BUF2)`	46000	Вывод остановлен.
3000	`RELEASE`	47000	`RELEASE, OUT BUF1`
4000	`ASSIGN(BUF3)`	52000	`ASSIGN(BUF2)`
5000	`RELEASE`	54500	Вывод остановлен.
6000	`ASSIGN` (ждать)	59000	`RELEASE, OUT BUF2`
8500	`BUF1` назначен, `OUT BUF2`	64000	`ASSIGN(BUF3)`
9500	`RELEASE`	65000	`RELEASE`
10500	`ASSIGN` (ждать)	66000	`ASSIGN(BUF1)`
16000	`BUF2` назначен, `OUT BUF3`	66500	`OUT BUF3`
23000	`RELEASE`	68000	`RELEASE`
23500	`OUT BUF1`	69000	Вычисления прекращены.
28000	`ASSIGN(BUF3)`	74000	`OUT BUF1`
31000	`OUT BUF2`	81500	Вывод остановлен.
35000	`RELEASE`		

Таким образом, всего потребовалось $81500u$; компьютер простаивал в промежутках 6000–8500, 10500–16000 и 69000–81500, т. е. всего $20500u$; устройство вывода было свободно в промежутках 0–1000, 46000–47000 и 54500–59000, т. е. всего $6500u$.

Для этой же программы создайте таблицу типа "время — действие", аналогичную приведенной выше, но при условии, что используются только *два* буфера.

10. [*21*] Выполните упр. 9, но только для *четырех* буферов.

11. [*21*] Выполните упр. 9, но только для *одного* буфера.

12. [*24*] Предположим, что алгоритм множественной буферизации, приведенный в тексте, используется для ввода с перфокарт. И пусть ввод должен быть прекращен сразу же, как только будет считана перфокарта, в колонке 80 которой содержится ".". Покажите, как следует модифицировать сопрограмму `CONTROL` (алгоритм B и программу B), чтобы ввод прекращался указанным образом.

13. [*20*] Пусть вывод выполняется с помощью алгоритмов буферизации. Какие команды нужно вставить в конце сопрограммы `COMPUTE`, приведенной в тексте раздела, чтобы гарантировать вывод всей информации из буферов?

▸ **14.** [*20*] Предположим, в вычислительной программе нет чередования действий `НАЗНАЧИТЬ` и `ОСВОБОДИТЬ`, а есть только последовательность действий ... `НАЗНАЧИТЬ` ... `НАЗНАЧИТЬ` ... `ОСВОБОДИТЬ` ... `ОСВОБОДИТЬ`. Какое влияние это окажет на алгоритмы, описанные в тексте раздела? Может ли это оказаться полезным?

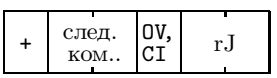
▸ **15.** [*22*] Напишите законченную программу для `MIX`, которая копирует 100 блоков с накопителя на магнитной ленте под номером 0 на аналогичное устройство номер 1, используя только три буфера. Программа должна работать настолько быстро, насколько это возможно.

16. [*29*] Сформулируйте алгоритм для зеленого-желтого-красного-фиолетового буферов, которые предложены на рис. 26, аналогичный алгоритмам множественной буферизации, приведенным в тексте раздела. Используйте три сопрограммы (одну для управления устройством ввода, другую — устройством вывода и третью — для вычислений).

17. [*40*] Переделайте алгоритм множественной буферизации для пула буферов; предусмотрите встроенные методы, которые не допускают замедления процесса из-за слишком большого объема опережающего ввода. Постарайтесь, по возможности, придать алгоритму красоту и изящество. Сравните свой метод с методами, в которых не используется пул, применяя их к реальным задачам.

▸ **18.** [*30*] Предлагаемое расширение машины `MIX` позволяет прерывать вычисления, как будет описано ниже. Ваша задача в этом упражнении — модифицировать приведенные в тексте раздела алгоритмы и программы A, R и B, чтобы вместо команд `JRED` в них использовались эти средства прерывания.

Новые возможности `MIX` включают 3 999 дополнительных ячеек памяти с адресами от -3999 до -0001. У этой машины есть два внутренних "состояния" — *нормальное* и *управляющее*. В нормальном состоянии ячейки с -3999 по -0001 недоступны и машина `MIX` работает, как обычно. Когда происходит "прерывание", вызванное условиями, о которых речь пойдет позже, в ячейки с -0009 по -0001 заносится содержимое регистров машины `MIX`: rA — в -0009; rI1–rI6 — в -0008–-0003; rX — в -0002, а rJ, состояние флага переполнения, флага сравнения и адрес следующей команды сохраняются в ячейке -0001 в следующем виде:



Когда машина входит в управляющее состояние, ячейка, которой передается управление, выбирается в зависимости от типа прерывания.

Ячейка −0010 играет роль часов: через каждые $1000u$ единиц времени число, содержащееся в этой ячейке, уменьшается на единицу; если в результате получается нуль, то происходит прерывание и управление передается в ячейку −0011.

Новая команда `MIX` "`INT`" (C = 5, F = 9) работает следующим образом. (a) В нормальном состоянии при прерывании управление передается ячейке −0012. (Таким образом, программист может вызвать прерывание, чтобы установить связь с управляющей программой; адрес `INT` значения не имеет, хотя управляющая программа может использовать его в информационном смысле, чтобы отличать один тип прерывания от другого.) (b) В управляющем состоянии все регистры `MIX` загружаются информацией из ячеек с −0009 по −0001, затем компьютер возвращается в нормальное состояние и возобновляет выполнение. Время выполнения команды `INT` в обоих случаях составляет $2u$.

Команда `IN`, `OUT` или `IOC`, выданная в *управляющем* состоянии, вызовет прерывание сразу же по окончании операции В/В. В этом случае управление будет передано в ячейку −(0020+ номер устройства).

В управляющем состоянии прерывания никогда не происходят. Любые условия, вызывающие прерывания, "сохраняются" до появления следующей операции `INT`, и прерывание происходит после выполнения одной команды в нормальном состоянии программы.

▸ **19.** [*M28*] Ввод-вывод небольших блоков данных с вращающегося устройства, например с магнитного диска, необходимо рассматривать особо. Предположим, программа работает с $n \geq 2$ последовательными блоками информации следующим образом. Блок k начинает вводиться в момент t_k, где $t_1 = 0$. Он назначается для обработки в момент $u_k \geq t_k + T$ и освобождает буфер в момент $v_k = u_k + C$. Диск совершает один оборот через каждые P единиц времени, и его считывающая головка пересекает начало нового блока через каждые L единиц времени; поэтому мы имеем $t_k \equiv (k-1)L$ (по модулю P). Так как обработка выполняется последовательно, имеем также $u_k \geq v_{k-1}$ при $1 < k \leq n$. Всего у нас N буферов; следовательно, $t_k \geq v_{k-N}$ при $N < k \leq n$.

Насколько большим должно быть N, чтобы время v_n завершения операции было минимальным из возможных, $T + C + (n-1)\max(L, C)$? Сформулируйте общее правило определения наименьшего такого N. Проиллюстрируйте свое правило для $L = 1$, $P = 100$, $T = .5$, $n = 100$ и (a) $C = .5$; (b) $C = 1.0$; (c) $C = 1.01$; (d) $C = 1.5$; (e) $C = 2.0$; (f) $C = 2.5$; (g) $C = 10.0$; (h) $C = 50.0$; (i) $C = 200.0$.

Поэтому в нашем примере имеем (a) $N = 1$; (b) $N = 2$; (c) $N = 3$, $c_N = 2.5$; (d) $N = 35$, $c_N = 51.5$; (e) $N = 51$, $c_N = 101.5$; (f) $N = 41$, $c_N = 102$; (g) $N = 11$, $c_N = 109.5$; (h) $N = 3$, $c_N = 149.5$; (i) $N = 2$, $c_N = 298.5$.

1.4.5. История и библиография

Большинство фундаментальных методов, описанных в разделе 1.4, было независимо разработано разными программистами, и подробная история возникновения их идей, видимо, никогда не будет достоверно известна. Поэтому сейчас мы просто попытаемся перечислить и оценить наиболее важные работы, которые внесли вклад в развитие этих идей.

Подпрограммы были первым средством экономии сил программистов. Еще в 19 веке Чарльз Бэббидж (Charles Babbage) предвидел возможность создания библиотеки программ для своей аналитической машины [см. *Charles Babbage and His Calculating Engines*, под редакцией Филиппа (Philip) и Эмили (Emily) Моррисон (Morrison) (Dover, 1961), 56]. И теперь можно сказать, что его мечта сбылась в 1944 году, когда Грэйс М. Хоппер (Grace M. Hopper) написала подпрограмму вычисления функции $\sin x$ для счетно-решающего устройства Mark I, установленного

в Гарвардском университете [см. *Mechanization of Thought Processes* (London: Nat. Phys. Lab., 1959), 164]. Но это еще были, в сущности, "открытые подпрограммы", т. е. такие подпрограммы, которые нужно вставлять в программу, а не связывать динамически. Управление машиной Бэббиджа, как и ткацким станком Жаккарда, осуществлялось с помощью набора перфокарт, а счетно-решающим устройством Mark I — с помощью бумажных перфолент. Таким образом, эти устройства существенно отличались от современных компьютеров с их хранящимися в памяти программами.

Связь с подпрограммами, реализуемая на машинах с хранящимися в памяти программами, где адрес, по которому нужно вернуть управление, передается в качестве параметра, была рассмотрена Германом Г. Голдстайном (Herman H. Goldstine) и Джоном фон Нейманом (John von Neumann) в их широкоизвестной монографии по программированию, написанной между 1946 и 1947 годами [см. работу фон Неймана *Collected Works* **5** (New York: Macmillan, 1963), 215–235]. В их программах главная программа отвечала за передачу параметров в тело подпрограммы, а не за передачу необходимой информации в регистры. В Англии А. М. Тьюринг (A. M. Turing) уже в 1945 году разработал аппаратное и программное обеспечение связи с подпрограммами [см. работу *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery* (Cambridge, Mass.: Harvard University, 1949), 87–90; B. E. Carpenter, R. W. Doran, editors, *A. M. Turing's ACE Report of 1946 and Other Papers* (Cambridge, Mass.: MIT Press, 1986), 35, 36, 76, 78, 79]. Способы применения и строение универсальной библиотеки подпрограмм — это главная тема первого учебника по компьютерному программированию M. V. Wilkes, D. J. Wheeler, S. Gill, *The Preparation of Programs for an Electronic Digital Computer*, 1st ed. (Reading, Mass.: Addison-Wesley, 1951) (Уилкс М., Уилер Д. и Гилл С. Составление программ для электронных счетных машин. — М.: Изд-во иностр. лит., 1953).

Слово "сопрограмма" придумал М. Э. Конвей (M. E. Conway) в 1958 году, после того как он разработал это понятие и впервые применил его при построении программы на языке ассемблера. Почти в то же время сопрограммы независимо изучали Дж. Эрдвинн (J. Erdwinn) и Дж. Мернер (J. Merner). Они написали статью "Bilateral Linkage"*, которую посчитали недостаточно интересной, чтобы быть достойной опубликования. К сожалению, до сегодняшнего дня ни один экземпляр этой статьи, похоже, не сохранился. Первое опубликованное объяснение понятия сопрограммы появилось значительно позже в статье Конвея "Design of a Separable Transition-Diagram Compiler", *CACM* **6** (1963), 396–408. На самом деле о примитивной форме связи сопрограмм уже кратко упоминалось в "советах по программированию" в первых публикациях, посвященных машине UNIVAC [*The Programmer* **1**, 2 (February, 1954), 4]. Удобное обозначение для сопрограмм в алголоподобных языках было введено в работе Dahl, Nygaard SIMULA I [*CACM* **9** (1966), 671–678]. Несколько прекрасных примеров сопрограмм (включая сопрограммы репликации) появилось в книге O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare *Structured Programming*, Chapter 3 (Дал У., Дейкстра Э., Хоор К. Структурное программирование / Пер. с англ. — М.: Мир, 1975 (сер. "Математическое обеспечение ЭВМ")). Первой программой-интерпретатором можно считать "универсальную машину Тьюринга", способную имитировать любые другие машины Тьюринга. Машины Тьюринга —

* "Двусторонняя связь". — *Прим. перев.*

это не реальные компьютеры, а теоретические конструкции, использующиеся для доказательства того, что некоторые задачи нельзя решить с помощью алгоритмов. Программы-интерпретаторы в традиционном смысле упоминал в 1946 году Джон Мочли (John Mauchly) в своих лекциях, которые он читал в Школе Мура (Moore School). Самыми известными из первых интерпретаторов, главным назначением которых было обеспечение удобных средств выполнения операций с плавающей точкой, были программы для машины Whirlwind I (разработанные Ч. В. Адамсом (C. W. Adams) и др.) и для машины Illiac I (разработанные Д. Дж. Уилером (D. J. Wheeler) и др.). Тьюринг также принимал участие в их разработке; интерпретирующие системы для компьютера Pilot ACE были написаны под его руководством. Более подробно о положении дел с интерпретаторами в начале 50-х годов говорится в статье J. M. Bennett, D. G. Prinz, M. L. Woods, "Interpretative Sub-routines", *Proc. ACM Nat. Conf.* (1952), 81–87 [см. также различные статьи в журнале *Proceedings of the Symposium on Automatic Programming for Digital Computers* (1954), издаваемом Департаментом морских исследований (Вашингтон, федеральный округ Колумбия).

Наиболее широкоиспользуемой программой-интерпретатором, вероятно, была "IBM 701 Speedcoding system" Джона Бэкуса (John Backus) [см. *JACM* **1** (1954), 4–6]. Этот интерпретатор был несколько модифицирован и искусно переписан для IBM 650 В. М. Волонтисом (V. M. Wolontis) и другими из компании Bell Telephone Laboratories; их программа "Bell Interpretive System" приобрела широкую популярность. Интерпретирующие системы IPL, которые были разработаны в начале 1956 года А. Ньювеллом (A. Newell), Дж. К. Шоу (J. C. Shaw) и Г. А. Саймоном (H. A. Simon) для совершенно других целей (см. раздел 2.6), широко использовались для обработки списков. Как уже упоминалось во введении к разделу 1.4.3, о современных способах применения интерпретаторов в компьютерной литературе, как правило, говорится "на ходу" [см. ссылки на статьи, перечисленные в этом разделе, в которых интерпретаторы обсуждаются более подробно].

Первая программа трассировки была разработана Стэнли Гиллом (Stanley Gill) в 1950 году [см. его интересную статью в журнале *Proceedings of the Royal Society of London*, series A, **206** (1951), 538–554]. В упомянутой выше книге Уилкса, Уилера и Гилла содержится несколько программ трассировки. Наверное, самая интересная из них — это подпрограмма C-10, написанная Д. Дж. Уилером, в которой предусмотрена возможность, позволяющая приостановить трассировку при входе в библиотечную подпрограмму, выполнить эту подпрограмму на полной скорости, а затем продолжить трассировку. Информация о программах трассировки довольно редко публикуется в общей компьютерной литературе. Это обусловлено, главным образом, тем, что данные методы по своей природе ориентированы на конкретную машину. Автору известна только одна ранняя работа на эту тему — статья H. V. Meek, "An Experimental Monitoring Routine for the IBM 705", *Proc. Western Joint Computer Conf.* (1956), 68–70. В ней обсуждается программа трассировки для машины, на которой было чрезвычайно трудно решить одну задачу. В настоящее время внимание к программам трассировки сместилось в сторону программ, обеспечивающих выборочный вывод результатов и оценок производительности программы; одна из лучших подобных систем была разработана Э. Сэттерсвейтом (E. Satterthwaite) и описана в журнале *Software Practice & Experience* **2** (1972), 197–217.

Буферизация первоначально выполнялась аппаратным способом, аналогично тому, как это делалось в программе 1.4.4–(3). Внутренняя буферная область, недо-

ступная для программиста, играла роль ячеек 2000–2099, и команды 1.4.4–(3) выполнялись неявно и незаметно, когда выдавалась команда ввода. В конце 40-х годов первые программисты UNIVAC разработали программные методы буферизации, которые особенно полезны для сортировки (см. раздел 5.5). Хороший обзор основных принципов В/В, преобладавших в 1952 году, можно найти в Трудах конференции Восточного компьютерного объединения, которая проводилась в 1952 году.

В компьютере DYSEAC [Alan L. Leiner, *JACM* **1** (1954), 57–81] была реализована идея непосредственной передачи информации между устройствами ввода-вывода и памятью во время работы программы, а затем прерывания программы по ее завершении. Из существования подобных систем следует, что для них были разработаны алгоритмы буферизации, но подробности не были опубликованы. В первой опубликованной работе о методах буферизации в том смысле, в котором мы их описали, представлен крайне сложный подход к этой проблеме [см. O. Mock, C. J. Swift, "Programmed Input-Output Buffering", *Proc. ACM Nat. Conf.* (1958), paper 19, и *JACM* **6** (1959), 145–151]. (Должен предупредить читателя о том, что в обеих статьях широко используется местный жаргон, для понимания которого придется потратить некоторое время, но вам помогут в этом другие статьи из журнала *JACM* **6**.) Система прерывания, которая позволяла осуществлять буферизацию ввода и вывода, была независимо разработана Э. В. Дейкстрой (E. W. Dijkstra) в 1957 и 1958 годах для компьютера $X-1$, созданного Б. Ж. Лупстрой (B. J. Loopstra) и К. С. Шолтеном (C. S. Scholten) [см. *Comp. J.* **2** (1959), 39–43]. В докторской диссертации Дейкстры, "Communication with an Automatic Computer"* (1958), обсуждаются методы буферизации, которые в данном случае были рассчитаны на очень длинные кольца буферов, в то время как в программах рассматривался, в основном, В/В на перфоленты и терминал. В каждом буфере содержался либо один символ, либо одно число. Впоследствии Дейкстра развил эти идеи и пришел к важному общему понятию *семафоров*, которые лежат в основе управления параллельными процессами любого вида, а не только вводом-выводом [см. *Programming Languages*, ed. by F. Genuys (Academic Press, 1968), 43–112; *BIT* **8** (1968), 174–186; *Acta Informatica* **1** (1971), 115–138]. В статье David E. Ferguson, "Input-Output Buffering and FORTRAN", *JACM* **7** (1960), 1–9, рассматриваются буферные кольца и приводится подробное описание простой буферизации для многих устройств одновременно.

Как представляется, примерно 1 000 команд — это разумный верхний предел сложности задач.

— ГЕРМАН ГОЛДСТАЙН (HERMAN GOLDSTINE) и ДЖОН ФОН НЕЙМАН (JOHN VON NEUMANN) (1946)

* "Связь с помощью автоматического компьютера". — *Прим. перев.*