1.4′. НЕКОТОРЫЕ ФУНДАМЕНТАЛЬНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

1.4.1′. Подпрограммы

Если некоторая задача выполняется в нескольких разных местах программы, то обычно не рекомендуется многократно повторять один и тот же фрагмент кода в каждом таком месте. Во избежание такого повторения, такой фрагмент кода (который называется *подпрограммой*) помещается только в одно место программы, а несколько дополнительных инструкций можно добавить для повторного запуска основной программы после завершения работы подпрограммы. Передача управления между подпрограммами и основными программами называется *связыванием подпрограмм*.

В каждом компьютере предусмотрен собственный способ эффективного связывания подпрограмм, который обычно включает несколько инструкций. Дальнейшее обсуждение касается компьютера `MMIX`, но аналогичные рассуждения верны и для связывания подпрограмм на других типах компьютеров.

Подпрограммы используются для экономии места в программе, а не времени, если не считать неявно сэкономленного времени в связи с ее уменьшенным размером, — например, на загрузку, либо более эффективное использование высокоскоростной памяти на компьютерах с разными типами памяти. Дополнительное время на вызов и выход подпрограммы обычно пренебрежимо мало, за исключением случаев, когда они находятся в глубоко вложенных циклах.

Подпрограммы обладают несколькими преимуществами. Они упрощают внешний вид большой и сложной программы, логически разбивают задачу на части, что обычно существенно упрощает отладку программы. Многие подпрограммы ценны еще и тем, что их могут использовать даже те пользователи, которые не участвовали в их создании.

Для большинства компьютеров предусмотрены большие библиотеки полезных подпрограмм, которые значительно упрощают программирование стандартных компьютерных приложений. Однако не следует считать это *единственным* предназначением подпрограмм. Подпрограммы не нужно рассматривать всего лишь как программы общего назначения. Специализированные подпрограммы имеют не меньшую важность, даже если предназначены только для использования в одной программе. В разд. 1.4.3′ содержится несколько типичных примеров таких ситуаций.

Простейшие подпрограммы имеют только один вход и один выход, как подпрограмма `MAXIMUM`, уже рассмотренная ранее (см. программу M в разд. 1.3.2′ и упр. 1.3.2′–3). Рассмотрим еще раз эту программу, слегка переделав ее так, чтобы максимальное значение находилось среди фиксированного числа ячеек, например среди 100:

```
* Поиск максимума среди X[1..100]
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SETL   kk,100*8  M1. Инициализировать.
       LDO    m,x0,kk
       JMP    1F
3H     LDO    xk,x0,kk  M3. Сравнить.
       CMP    t,xk,m                                        (1)
       PBNP   t,5F
4H     SET    m,xk      M4. Изменить m.
1H     SR     j,kk,3
5H     SUB    kk,kk,8   M5. Уменьшить k.
       PBP    kk,3B     M2. Все проверены?
6H     POP    2,0       Возврат в основную программу. ∎
```

Предполагается, что эта подпрограмма является частью более крупной программы, в которой символ `t` обозначает регистр \$255, а символ `x0` обозначает глобальный регистр так, что $X[k]$ находится в позиции `x0`$+8k$. В этой более крупной программе инструкция "`PUSHJ $1,Max100`" устанавливает для регистра \$1 текущее максимальное значение среди $\{X[1], \ldots, X[100]\}$, а позиция максимума заносится в регистр \$2. Связывание в этом случае достигается с помощью инструкции `PUSHJ`, которая вызывает подрограмму, и с помощью инструкции "`POP 2,0`" в конце подпрограммы. Как будет показано ниже, эти инструкции `MMIX` вызывают перенумерацию локальных регистров во время работы подпрограммы. Более того, инструкция `PUSHJ` вставляет адрес возврата в специальный регистр rJ, а инструкция `POP` переходит в эту позицию.

Связывание подпрограмм можно выполнить более простым способом с использованием инструкции `GO` вместо инструкций проталкивания и выталкивания. Например, для этого можно использовать следующий код вместо (1):

```
* Поиск максимума среди X[1..100]
j GREG ;m GREG ;kk GREG ;xk GREG
         GREG   @          Базовый адрес.
GoMax100 SETL   kk,100*8   M1. Инициализировать.
         LDO    m,x0,kk                                    (2)
         JMP    1F
3H       ...               (Продолжить, как в (1))
         PBP    kk,3B      M2. Все проверены?
6H       GO     kk,$0,0    Возврат в основную программу. ∎
```

Теперь инструкция "`GO $0,GoMax100`" передает управление подпрограмме, заменяя адрес следующей инструкции в регистре \$0; последующая операция "`GO kk,$0,0`" в конце подпрограммы вернется к этому адресу. В этом случае максимальное значение появится в глобальном регистре `m`, а его позиция будет в глобальном регистре `j`. Два дополнительных глобальных регистра, `kk` и `xk`, также используются данной подпрограммой. Более того, инструкция "`GREG @`" предоставляет базовый адрес так, что можно перейти к `GoMax100` с помощью одной инструкции. В противном случае потребуется двухстадийная последовательность "`GETA $0,GoMax100; GO $0,$0,0`". Связывание подпрограмм типа (2) обычно используется на компьютерах без встроенного механизма стека регистров.

Не трудно сделать *количественную* оценку сэкономленного кода и затраченного времени при использовании подпрограмм. Предположим, что для фрагмента кода требуется k тетрабайт и он используется в m местах программы. Для преобразования этого фрагмента в подпрограмму нужно вставить инструкцию `PUSHJ` или `GO` в каждое из m мест вызова подпрограммы, а также использовать инструкцию `POP` или `GO` для возврата управления.

В целом это дает $m + k + 1$ тетрабайт, вместо mk, а потому сэкономленный код равен

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

Если k равно 1 или m равно 1, то очевидно никакой экономии вообще не будет. Если k равно 2, то для получения выгоды m должно быть больше 3.

Затраченное время равняется времени выполнения инструкций `PUSHJ`, `POP` и/или `GO` в процессе связывания. Если подпрограмма вызывается t раз во время выполнения программы, а время выполнения программы определяется приближениями в табл. 1.3.1′–1, то дополнительные затраты равны $4tv$ в случае (1) или $6tv$ в случае (2).

Эти оценки нужно использовать осторожно, потому что они получены для идеализированной ситуации. Многие подпрограммы нельзя вызвать с помощью одной инструкции `PUSHJ` или `GO`. С одной стороны, если фрагмент кода реплицируется во многих местах программы без применения подпрограмм, то в каждом таком случае его можно настроить так, чтобы воспользоваться преимуществами особенностей именно этого места программы, в которой находится данный фрагмент кода. С другой стороны, при использовании подпрограммы фрагмент кода должен иметь довольно общий вид, а потому в него часто включают несколько дополнительных инструкций.

Если подпрограмма создается для обработки общего случая, то она выражается с помощью *параметров*. Параметрами называются величины, которые управляют действиями подпрограммы. Их значения могут меняться при разных вызовах подпрограммы.

Код во внешней программе, который передает управление подпрограмме и запускает ее, называется *вызывающей последовательностью*. Значения параметров, которые передаются при вызове подпрограммы, называются *аргументами*. В подпрограмме `GoMax100` вызывающей последовательностью является “`GO $0,GoMax100`”, но при использовании аргументов вызывающая последовательность обычно длиннее.

Например, попробуем обобщить подпрограмму (2), чтобы она находила максимум среди первых n элементов массива для *произвольной* константы n, помещая n в поток инструкций с двухстадийной вызывающей последовательностью

$$\texttt{GO \$0,GoMax;}\quad \texttt{TETRA}\ n. \tag{4}$$

Подпрограмма GoMax в этом случае будет иметь вид:

```
* Поиск максимума среди X[1..100]
j GREG ;m GREG ;kk GREG ;xk GREG
       GREG   @         Базовый адрес.
GoMax  LDT    kk,$0,0   Извлечь аргумент.
       SL     kk,kk,3
       LDO    m,x0,kk
       JMP    1F
3H     ...              (Продолжить, как в (1))
       PBP    kk,3B
6H     GO     kk,$0,4   Возврат в вызывающую программу. ∎
```
(5)

Параметр n рекомендуется поместить в регистр. Тогда двухстадийная вызывающая последовательность будет иметь вид:

```
SET $1,n;  GO $0,GoMax
```
(6)

и соответствующая подпрограмма будет выглядеть так:

```
GoMax  SL     kk,$1,3   Извлечь аргумент.
       LDO    m,x0,kk
       ...
6H     GO     kk,$0,0   Возврат в вызывающую программу. ∎
```
(7)

Этот вариант быстрее, чем (5), и позволяет динамически изменять n без изменения потока инструкций.

Обратите внимание, что адрес элемента массива $X[0]$ также является параметром подпрограмм (1), (2), (5) и (7). Операция помещения этого адреса в регистр x0 может рассматриваться как часть вызывающей последовательности в тех случаях, когда массив меняется при каждом вызове.

Если вызывающая последовательность занимает c тетрабайт памяти, то формула (3) для сэкономленного пространства принимает вид:

$$(m-1)\,(k-c) - \text{константа} \tag{8}$$

и немного увеличивается время, затраченное на связывание подпрограммы.

Дополнительное уточнение приведенных выше формул потребуется, поскольку нужно сохранять и восстанавливать содержимое некоторых регистров. Например, в подпрограмме GoMax нужно помнить, что инструкция “SET $1,$n$; GO $0,GoMax” не только вычисляет максимальное значение в регистре m и его положение в регистре j, но также изменяет значения глобальных регистров kk и xk. Подпрограммы (2), (5) и (7) созданы с неявным предположением, что регистры kk и xk используются только для данной подпрограммы, но в большинстве компьютеров не так много регистров.

Даже в MMIX регистры быстро закончатся, если запустить большое количество подпрограмм. Поэтому подпрограмму (7) нужно подкорректировать так, чтобы она работала с kk $\equiv$ \$2 и xk $\equiv$ \$3 без затирания полезного содержимого этих регистров.

Это достигается следующей корректировкой подпрограммы

```
j GREG ;m GREG ;kk IS $2 ;xk IS $3
       GREG   @          Базовый адрес.
GoMax  STO    kk,Tempkk  Сохранить предыдущее содержимое регистра.
       STO    xk,Tempxk
       SL     kk,$1,3    Извлечь аргумент.
       LDO    m,x0,kk
       ...
       LDO    kk,Tempkk  Восстановить предыдущее содержимое регистра.
       LDO    xk,Tempxk
6H     GO     $0,$0,0    Возврат в вызывающую программу. ▮
```
(9)

и установкой двух октабайтов `Tempkk` и `Tempxk` в сегменте данных. Конечно, это изменение увеличивает накладные расходы при каждом использовании подпрограммы.

Подпрограмму можно рассматривать как *расширение* машинного языка данного компьютера. Например, если подпрограмма `GoMax` находится в памяти, то таким образом в нашем распоряжении оказывается машинная инструкция (а именно, "`GO $0,GoMax`"), которая способна отыскать максимум. Здесь важно так же тщательно определить каждую подпрограмму, как определяются операторы машинного языка. Программисту всегда настоятельно рекомендуется указывать все характеристики подпрограммы, даже если никто кроме него не будет пользоваться этой подпрограммой или ее спецификацией. В случае подпрограммы `GoMax` с определением (7) или (9) ее характеристики имеют следующий вид:

Вызывающая последовательность:	`GO $0,GoMax`.	
Условия входа:	$\$1 = n \geq 1$; `x0` = address of $X[0]$.	(10)
Условия выхода:	$\mathtt{m} = \max_{1\leq k\leq n} X[k] = X[\mathtt{j}]$.	

В спецификации следует упомянуть все изменения тех величин, которые являются внешними для подпрограммы. Если регистры `kk` и `xk` не считаются "частными" для варианта (7) подпрограммы `GoMax`, следует учесть, что эти регистры вовлекаются в процесс выполнения подпрограммы, как часть условий входа. Подпрограмма также изменяет регистр `t`, а именно регистр \$255, но этот регистр обычно используется для временных значений, поэтому не нужно указывать его явно.

Рассмотрим теперь *многократные входы* в подпрограммы. Предположим, имеется программа, в которой используется общая подпрограмма `GoMax`, но требуется применить специализированную подпрограмму `GoMax100`, в которой $n = 100$. Их можно скомбинировать таким образом:97.02.24

```
GoMax100 SET $1,100  Первый вход.
GoMax    ...         Второй вход; продолжить, как в (7) или (9). ▮
```
(11)

Можно также добавить *третий* вход, например `GoMax50`, используя код

```
GoMax50 SET $1,50;  JMP GoMax
```

в некотором месте.

Подпрограмма может иметь *многократные выходы*, т.е. возвращать значения в одно из нескольких разных мест, в зависимости от заданных условий.

Например, можно расширить подпрограмму (11) с помощью предположения, что параметр верхней границы задан в глобальном регистре b. Теперь предполагается, что выход подпрограммы произойдет в одном и двух тетрабайтов, которые располагаются вслед за вызывающей инструкцией GO:

Для произвольного n	Для $n = 100$
SET $1,$n$; GO $0,GoMax	GO $0,GoMax100
Выход здесь, если m ≤ 0 или m $\geq$ b.	Выход здесь, если m ≤ 0 или m $\geq$ b.
Выход здесь, если $0 <$ m $<$ b.	Выход здесь, если $0 <$ m $<$ b.

(Иначе говоря, тетрабайт после GO пропускается, когда максимальное значение положительно и меньше верхней границы. Такая подпрограмма полезна в программах, где часто нужно принять определенное решение после вычисления максимального значения.) Ее реализация выглядит достаточно просто:

```
* Поиск максимума X[1..n] с проверкой границ
j GREG ;m GREG ;kk GREG ;xk GREG
         GREG   @         Базовый адрес.
GoMax100 SET    $1,100    Вход для n = 100.
GoMax    SL     kk,$1,3   Вход для произвольного n.
         LDO    m,x0,kk
         JMP    1F
3H       ...              (Продолжить, как в (1)).
         PBP    kk,3B
         BNP    m,1F      Переход, если m ≤ 0.
         CMP    kk,m,b
         BN     kk,2F     Переход, если m < b.
1H       GO     kk,$0,0   Первый выход, если m ≤ 0 или m ≥ b.
2H       GO     kk,$0,4   В противном случае второй выход. ∎
```
(12)

Обратите внимание, что в этой программе комбинируется технология связывания потока инструкций (5) с технологией установки регистра (7). Строго говоря, положением выхода подпрограммы является параметр. Следовательно, положения нескольких выходов могут задаваться в виде аргументов. Если подпрограмма постоянно осуществляет доступ к своим параметрам, то соответствующий аргумент лучше всего передавать в регистре, но если аргумент является константой и используется редко, то лучше хранить его в потоке инструкций.

Подпрограммы могут вызывать другие подпрограммы. Действительно, в очень сложных программах часто вызовы подпрограмм могут быть вложены на глубину более пяти уровней. Единственным ограничением в случаях связывания подпрограмм на основе GO является то, что все адреса и регистры временного хранения должны быть разными, таким образом, не допускается вызов подпрограммой любой другой подпрограммы, которая (прямо или косвенно) вызывает первую подпрограмму. Рассмотрим в качестве примера следующий сценарий.

```
[Основная программа]   | [Подпрограмма A] | [Подпрограмма B] | [Подпрограмма C]
                       |                  |                  |
                       | A                | B                | C
      ⋮                |     ⋮            |     ⋮            |     ⋮
   GO $0,A             |    GO $1,B       |    GO $2,C       |    GO $0,A
      ⋮                |     ⋮            |     ⋮            |     ⋮
                       |    GO $0,$0,0    |    GO $1,$1,0    |    GO $2,$2,0
```
(13)

Если основная программы вызывает подпрограмму A, которая вызывает подпрограмму B, которая вызывает подпрограмму C, а потом C вызывает подпрограмму A, то адрес в $0, относящийся к основной программе, уничтожается, а потому обрывается путь возврата в основную программу.

Использование стековой памяти. Хотя рекурсивное использование подпрограмм, как в примере (13), редко возникает в простых программах, оно гораздо чаще встречается в исходной структуре важных приложений. К счастью, существует простой и прямолинейный способ исключения взаимного вредного влияния между вызовами подпрограмм за счет сохранения их локальных переменных в *стеке*.

Например, можно задать глобальный регистр sp ("stack pointer" — указатель стека) и применить инструкцию GO $0,Sub для вызова каждой программы. Если код подпрограммы имеет вид

```
Sub  STO  $0,sp,0
     ADD  sp,sp,8
     ...
     SUB  sp,sp,8
     LDO  $0,sp,0
     GO   $0,$0,0
```
(14)

то регистр $0 всегда будет содержать соответствующий адрес возврата, а проблема (13) будет исключена. (Сначала в sp заносится адрес в сегменте данных, а затем все другие необходимые адреса памяти.) Более того, инструкции STO/ADD и SUB/LDO в (14) можно опустить, если Sub является так называемой *концевой подпрограммой*, — которая не вызывает других подпрограмм.

Стек можно использовать для хранения параметров и других локальных переменных, кроме адресов возврата в (14). Предположим, что в подпрограмме Sub, кроме адреса возврата, используется 20 октабайт локальных данных. В этой ситуации можно использовать следующую схему:

```
Sub  STO  fp,sp,0   Сохранить прежний указатель стекового фрейма.
     SET  fp,sp     Установить новый указатель стекового фрейма.
     INCL sp,8*22   Передвинуть указатель стека.
     STO  $0,fp,8   Сохранить адрес возврата.
     ...                                                          (15)
     LDO  $0,fp,8   Восстановить адрес возврата.
     SET  sp,fp     Восстановить указатель стека.
     LDO  fp,sp,0   Восстановить указатель стекового фрейма.
     GO   $0,$0,0   Вернуться в вызывающую программу. ▌
```

Здесь fp является глобальным регистром, который называется *указателем стекового фрейма*.

В отмеченной многоточием части подпрограммы локальное значение k эквивалентно октабайту по адресу `fp` $+ 8k + 8$ для $1 \le k \le 20$. Инструкции в начале "проталкивают" локальные значения в "верх" стека, а инструкции в конце "выталкивают" их из стека, оставляя стек в том же состоянии, которое он имел во время входа подпрограммы.

Использование стека регистров. Связывание подпрограмм на основе `GO` столь подробно рассматривалось здесь, поскольку многие компьютеры не имеют более удачных альтернативных вариантов связывания подпрограмм.

Но в `MMIX` встроены инструкции `PUSHJ` и `POP`, которые гораздо более эффективно управляют связыванием подпрограмм и позволяют сократить многие накладные расходы, которые присущи схемам (9) и (15). Эти инструкции позволяют сохранять большинство параметров и локальных переменных в регистрах вместо хранения их в стековой памяти и повторной загрузки. С помощью инструкций `PUSHJ` и `POP` большая часть рутинной работы со стеком автоматически выполняется компьютером.

Основная идея стека регистров чрезвычайно проста, если вспомнить принцип работы стека, как такового. `MMIX` имеет *стек регистров* из октабайтов $S[0]$, $S[1]$, $\ldots$, $S[\tau-1]$ для некоторого числа $\tau \ge 0$. Самые верхние L октабайтов этого стека (а именно $S[\tau-L]$, $S[\tau-L+1]$, $\ldots$, $S[\tau-1]$) являются *локальными регистрами* \$0, \$1, $\ldots$, \$(L−1), а другие $\tau-L$ октабайт стека пока недоступны программе и будут называться "протолкнутыми вниз". Текущее количество локальных регистров, L, хранится в специальном регистре rL компьютера `MMIX`, хотя программисту совсем не обязательно это знать. В исходном состоянии $L = 2$, $\tau = 2$, а локальные регистры \$0 и \$1 представляют командную строку, как в программе 1.3.2′H.

`MMIX` также имеет *глобальные регистры*, а именно \$$G$, \$($G+1$), $\ldots$, \$255. Значение G хранится в специальном регистре rG, причем всегда $0 \le L \le G \le 255$. (На самом деле также всегда выполняется условие $G \ge 32$.) Глобальные регистры *не* являются частью стека регистров.

Регистры, которые не являются ни локальными, ни глобальными, называются *маргинальными*. Эти регистры, а именно \$$L$, \$($L+1$), $\ldots$, \$($G-1$), имеют нулевое значение всякий раз, когда они используются как входные операнды инструкции `MMIX`.

Стек регистров возрастает, если маргинальному регистру присваивается значение. Этот маргинальный регистр становится локальным, как и все маргинальные регистры с меньшими номерами. Например, если в данный момент используется восемь локальных регистров, то инструкция `ADD $10,$20,5` приводит к тому, что регистры \$8, \$9, и \$10 становятся локальными. Точнее говоря, если rL = 8, то инструкция `ADD $10,$20,5` задает \$8 ← 0, \$9 ← 0, \$10 ← 5, и rL ← 11. (А регистр \$20 остается маргинальным.)

Если \$X является локальным регистром, то инструкция `PUSHJ $X,Sub` уменьшает количество локальных регистров и и изменяет их порядковые номера: прежние локальные регистры \$(X + 1), \$(X + 2), $\ldots$, \$($L-1$) теперь называются \$0, \$1, $\ldots$, \$($L-$X-2) внутри подпрограммы, а значение L уменьшается на X + 1. Таким образом, стек регистра остается неизменным, но X + 1 его элементов становятся недоступными. Подпрограмма не может повредить эти элементы и получает X + 1 новых маргинальных регистров.

Если X $\geq G$, а \$X является глобальным регистром, то результат выполнения инструкции `PUSHJ $X,Sub` аналогичен, но новый элемент помещается в стек регистров и потом $L+1$ регистров проталкиваются вниз вместо X + 1. В данном случае значение L равно нулю в начале подпрограммы, все прежние локальные регистры проталкиваются вниз, а подпрограмма начинается с чистого листа.

Стек регистров сжимается, только если задается инструкция `POP` или программа явно уменьшает количество локальных регистров с помощью инструкции `PUT rL,5`. Цель `POP X,YZ` состоит в том, чтобы сделать снова доступными элементы, проталкиваемые вниз самой последней инструкцией `PUSHJ`, как прежде, и удалить из стека регистров более неиспользуемые элементы. Вообще, поле X инструкции `POP` содержит количество значений, "возвращаемых" подпрограммой, если X $\leq L$. Если X > 0, то возвращаемое значение равно \$(X−1). Из регистра удаляется этот элемент и все элементы выше него, а возвращаемое значение помещается в адрес, указанный командой `PUSHJ`, которая вызывала данную подпрограмму. Поведение инструкции `POP` аналогично, если X $> L$, но в данном случае стек регистров остается неизменным и нулевое значение помещается по адресу, указанному командой `PUSHJ`.

Описанные выше правила могут показаться сложными, поскольку на практике могут иметь место разные ситуации. Однако несколько примеров могут прояснить их. Предположим, что есть подпрограмма `A` и в ней нужно вызвать подпрограмму `B`. Допустим, что подпрограмма `A` имеет 5 локальных регистров, которые не должны быть доступны подпрограмме `B`. Этими регистрами являются \$0, \$1, \$2, \$3 и \$4. Зарезервируем следующий регистр \$5 для основного результата подпрограммы `B`. Если `B` имеет три параметра, то \$6 ← arg0, \$7 ← arg1 и \$8 ← arg2, и после команды `PUSHJ $5,B` вызывается подпрограмма `B` и аргументы помещаются в \$0, \$1 и \$2.

Если `B` не возвращает результат, то ее выполнение завершается командой `POP 0,YZ`, которая восстанавливает \$0, \$1, \$2, \$3 и \$4 к исходным значениям и задает $L \leftarrow 5$.

Если `B` возвращает единственный результат x, то x помещается в \$0 и ее выполнение завершается командой `POP 1,YZ`, которая восстанавливает \$0, \$1, \$2, \$3, и \$4 к прежним значениям и задает \$5 ← x и $L \leftarrow 6$.

Если `B` возвращает два результата x и a, то основной результат x помещается в \$1 и вспомогательный результат a помещается в \$0. Затем `POP 2,YZ` восстанавливает регистры от \$0 до \$4 и задает \$5 ← x, \$6 ← a, $L \leftarrow 7$. Аналогично, если `B` возвращает десять результатов $(x, a_0, \ldots, a_8)$, то основной результат x помещается в \$9 и остальные результаты в первые девять регистров: \$0 ← a_0, \$1 ← a_1, ..., \$8 ← a_8. Затем `POP 10,YZ` восстанавливает регистры от \$0 до \$4 и задает \$5 ← x, \$6 ← a_0, ..., \$14 ← a_8. (Забавная перестановка регистров, которая возникает, если возвращается два или более результата, может показаться довольно странной на первый взгляд. Однако она имеет смысл, поскольку оставляет стек регистров без изменений, за исключением основного результата. Например, если подпрограмма `B` имеет аргументы arg0, arg1 и arg2 в регистрах \$6, \$7 и \$8 по окончании своей работы, то она может сохранить их в виде вспомогательных результатов в регистрах \$0, \$1 и \$2, а потом выполнить инструкцию `POP 4,YZ`.)

Поле YZ инструкции `POP` обычно нулевое, но, вообще говоря, инструкция `POP X,YZ` возвращается к инструкции, которая располагается через YZ + 1 тетрабайт после `PUSHJ`, которая вызвала текущую подпрограмму. Это обобщение полезно для

подпрограмм с несколькими выходами. Точнее говоря, PUSHJ по адресу @ задает специальный регистр rJ для @ + 4 до перехода к подпрограмме, а инструкция POP затем возвращается к адресу rJ + 4YZ.

Теперь можно переписать созданные ранее программы со связыванием подпрограмм на основе инструкции GO так, чтобы в них использовалось связывание на основе инструкций PUSH/POP. Например, подпрограмма поиска максимума (12) с двумя входами и двумя выходами примет следующий вид, если в ней используется механизм стека регистров MMIX:

```
* Поиск максимума среди X[1..n] с проверкой границ.
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SET    $0,100    Вход для n = 100.
Max    SL     kk,$0,3   Вход для произвольного n.
       LDO    m,x0,kk
       JMP    1F                                        (16)
       ...              (Продолжить, как в (12)).
       BN     kk,2F
1H     POP    2,0       Первый выход, если max ≤ 0 или max ≥ b.
2H     POP    2,1       В противном случае второй выход. ∎
```

Для произвольного n	Для $n = 100$
SET $A,$n$; PUSHJ $R,Max (A = R+1)	PUSHJ $R,Max100
Выход здесь, если $R ≤ 0 или $R ≥ b.	Выход здесь, если $R ≤ 0 или $R ≥ b.
Выход здесь, если 0 < $R < b.	Выход здесь, если 0 < $R < b.

Локальный регистр результата $R в PUSHJ в этой вызывающей последовательности является произвольным, в зависимости от количества локальных переменных, которые вызывающая программа хочет сохранить. Локальный регистр аргумента $A переходит в $(R + 1). После этого вызова $R будет содержать основной результат (максимальное значение) и $A будет содержать вспомогательный результат (индекс элемента массива с этим максимумом). При наличии нескольких аргументов и/или вспомогательных результатов они именуются A0, A1, ..., причем предполагается, что A0 = R+1, A1 = R+2, ... , если используются вызывающие последовательности с PUSH/POP.

Сравнение (12) и (16) показывает, что у (16) есть небольшие преимущества: в новом варианте необязательно выделять глобальные регистры для j, m, kk и xk, а также необязательно использовать глобальный базовый регистр для адреса команды GO. (Напомним из раздела 1.3.1′, что GO имеет абсолютный, а PUSHJ — относительный адрес.) Инструкция GO немного медленнее, чем PUSHJ и POP, согласно табл. 1.3.1′–1, хотя высокоскоростные реализации MMIX могут реализовать POP более эффективно. Программы (12) и (16) имеют одинаковый размер.

Преимущества связывания на основе PUSH/POP по сравнению со связыванием на основе GO начинают проявляться при работе с *неконцевыми* подпрограммами (а именно, подпрограммами, которые вызывают другие подпрограммы, возможно, даже самих себя). Код (14) на основе GO можно заменить так:

```
Sub  GET  retadd,rJ
     ...
     PUT  rJ,retadd                                     (17)
     POP  X,0
```

где `retadd` является локальным регистром. (Например, `retadd` может быть равным $5; его номер регистра обычно больше или равен количеству возвращаемых результатов X, потому инструкция `POP` автоматически удалит его из стека регистров.) Таким образом исключаются затратные ссылки на память в (14).

Неконцевая подпрограмма со многими локальными переменными и/или параметрами гораздо эффективнее со стеком регистров, чем со стековой памятью (15), поскольку часто вычисления непосредственно выполняются в регистрах. Однако следует отметить, что стек регистров `MMIX` применяется только к локальным переменным, которые являются *скалярами*, а не к локальным переменным-массивам, для доступа к элементам которых требуется вычислять адреса. Для подпрограмм с нескалярными локальными переменными следует использовать схему (15) для всех таких переменных, а для скаляров применять стек регистров.

Оба подхода можно использовать одновременно, обновляя `fp` и `sp` только подпрограммами, в которых нужно применить стек регистров.

Если стек регистров становится чрезвычайно большим, то `MMIX` автоматически сохранит самые нижние элементы в сегменте памяти стека, используя скрытую процедуру, которая описывается в разделе 1.4.3′. (Напомним из раздела 1.3.2′, что этот сегмент памяти стека начинается с адреса `#6000000000000000`.) `MMIX` сохраняет элементы стека регистра в памяти также, когда команда `SAVE` сохраняет весь текущий контекст программы. Сохраненные элементы стека автоматически восстанавливаются из памяти при выполнении команды `POP` или команды `UNSAVE`, которая восстанавливает сохраненный контекст. Но в большинстве случаев `MMIX` способен проталкивать и выталкивать локальные регистры без фактического доступа к памяти и без фактического изменения содержимого очень многих внутренних регистров компьютера.

Стеки имеют много других способов применения в компьютерных программах. Их основные свойства рассматриваются в разделе 2.2.1. Более подробно вложенные подпрограммы и рекурсивные процедуры рассматриваются в разделе 2.3 при обсуждении операций с деревьями. В главе 8 подробно рассматривается рекурсия.

***Компоненты языка ассемблера.** Язык ассемблера `MMIX` поддерживает создание подпрограмм тремя способами, которые не упоминались в разделе 1.3.2′. Наиболее важной является операция `PREFIX`, которая упрощает определение “частных” символов, не взаимодействующих с символами, определенными в других местах большой программы. Основная идея состоит в том, что символ может иметь структурированный вид, например `Sub:X` (что значит символ `X` подпрограммы `Sub`), возможно, передаваемый на несколько уровней в виде `Lib:Sub:X` (что значит символ `X` подпрограммы `Sub` в библиотеке `Lib`).

Структурированные символы используют слегка расширенное правило 1 языка `MMIXAL` из раздела 1.3.2′, которое позволяет рассматривать двоеточие ‘`:`’ как “букву” для конструирования символов. Каждый символ, который не начинается с двоеточия, неявно расширяется размещением перед ним *текущего префикса*. Текущий префикс имеет вид ‘`:`’, но пользователь может изменить его с помощью команды

PREFIX. Например:

```
ADD     x,y,z              означает  ADD :x,:y,:z
PREFIX  Foo:               текущим префиксом является  :Foo:
ADD     x,y,z              означает  ADD :Foo:x,:Foo:y,:Foo:z
PREFIX  Bar:               текущим префиксом является  :Foo:Bar:
ADD     :x,y,:z            означает  ADD :x,:Foo:Bar:y,:z
PREFIX  :                  текущим префиксом является  :
ADD     x,Foo:Bar:y,Foo:z  означает  ADD :x,:Foo:Bar:y,:Foo:z
```

Один их способов использования этой идеи является замена первых строк в (16) на

```
        PREFIX Max:
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
x0 IS :x0 ;b IS :b ;t IS :t       Внешние символы.
:Max100 SET    $0,100    Вход для n = 100.
:Max    SL     kk,$0,3   Вход для произвольного n.
        LDO    m,x0,kk
        JMP    1F
        ...               (Продолжить, как в (16)).
```
(18)

и включение “PREFIX :” в конце. Далее символы j, m, kk и xk можно использовать в остальной части программы или в определении других подпрограмм. Другие примеры использования префиксов рассматриваются в разделе 1.4.3′.

В MMIXAL также предусмотрен псевдооператор LOCAL. Команда ассемблера “LOCAL $40” означает, например, что сообщение об ошибке появится в конце сборки, если команды GREG выделят так много регистров, что $40 будет глобальным. (Этот компонент необходим только, если подпрограмма использует более 32 локальных регистров, поскольку выражение “LOCAL $31” всегда неявно истинно.)

Предусмотрен также третий компонент поддержки подпрограммы, BSPEC ... ESPEC. Это позволяет передавать информацию объектному файлу так, чтобы процедуры отладки и другие системные программы знали о типах связывания, которые используются каждой подпрограммой. Этот компонент обсуждается в документации *MMIXware* и представляет особый интерес при компилировании программ.

Стратегические соображения. В специализированных подпрограммах инструкции GREG можно использовать более свободно, чтобы заполнить глобальные регистры основными константами для ускорения работы программы. В таких случаях требуется сравнительно небольшое количество локальных регистров, если только подпрограммы не используются рекурсивно.

Но если десятки или сотни подпрограмм общего назначения создаются для включения в крупную библиотеку, которая может использоваться произвольной пользовательской программой, то, очевидно, нельзя для каждой такой подпрограммы выделять большое количество глобальных регистров. Даже по одной глобальной переменной на одну подпрограмму может оказаться чересчур много.

Таким образом, инструкцию GREG следует применять часто только при малом и очень экономно при большом количестве подпрограмм. В последнем случае без значительной утраты эффективности можно использовать локальные переменные.

В завершение раздела кратко рассмотрим процесс создания сложной и длинной программы. Какой тип подпрограмм нужно использовать в ней? Какие вызывающие последовательности использовать? Успешный способ решения этих задач заключается в использовании следующей итеративной процедуры:

Этап 0 (Исходная идея). Сначала нужно выбрать общий план программы.

Этап 1 (Грубый эскиз программы). Начните с создания "внешних уровней" программы на любом удобном языке. Один из систематических способов создания эскиза прекрасно описан в книге E. W. Dijkstra, *Structured Programming* (Academic Press, 1972), Chapter 1, и в статье N. Wirth, *CACM* **14** (1971), 221– 227. Сначала вся программа разбивается на небольшое количество частей, которые временно можно рассматривать как подпрограммы, хотя они вызываются всего по одному разу. Эти части успешно разбиваются на все меньшие и меньшие части, соответственно выполняющие все более простую работу. Всякий раз, когда появляется некое похожее или повторяющееся вычисление, следует определить подпрограмму (теперь уже настоящую), которая выполнит это вычисление. На этом этапе код подпрограммы еще не создается, а продолжается работа с основной программой в предположении, что подпрограмма уже выполнила свою задачу. Наконец, после создания эскиза основной программы можно приступать к созданию подпрограмм, начиная с самых сложных и деля их на меньшие подпрограммы, и т.д. Продолжая в этой манере получаем список подпрограмм. Фактическая функция каждой подпрограммы уже, вероятно, могла измениться несколько раз так, что первые наброски эскиза могут оказаться неверными. Однако это не представляет большой проблемы, поскольку идет работа над общим эскизом. В результате этой работы создается представление о каждой вызываемой подпрограмме и ее общем предназначении. Следует учесть возможность даже минимального обобщения каждой подпрограммы.

Этап 2 (Первая работающая программа). На следующем этапе следует двигаться в противоположном направлении от этапа 1. Теперь можно приступать к созданию программы на компьютерном языке, например на языке `MMIXAL` или `PL/MMIX`, или — что более вероятно — языке более высокого уровня. Начнем с программирования подпрограмм самого низкого уровня, а код основной программы создадим в самую последнюю очередь. По мере возможности не рекомендуется создавать любые инструкции, которые вызывают подпрограмму до ее кодирования. (На этапе 1 мы поступали наоборот, т.е. не рассматривали подпрограмму до тех пор, пока не будут созданы все ее вызовы.)

По мере создания все большего числа подпрограмм уверенность программиста растет, поскольку по мере программирования растет вычислительная мощь компьютера. После создания кода отдельной подпрограммы следует немедленно подготовить полное описание выполняемых ею действий, а также ее вызывающих последовательностей, как в (10). Важно убедиться, что глобальные переменные не используются одновременно для двух конфликтующих целей. При подготовке эскиза на этапе 1 об этих проблемах не стоит беспокоиться.

Этап 3 (Повторная проверка). Результатом этапа 2 будет практически готовая работающая программа, но ее все-таки нужно попробовать улучшить. Для этого рекомендуется пойти в обратном направлении, изучая *все* места вызова каждой подпрограммы. Возможно подпрограмму нужно увеличить для выполнения более

общих действий, которые выполняются участками кода до и после вызова подпрограммы. Может быть, несколько подпрограмм лучше было бы объединить в одной подпрограмме. Иногда подпрограмма вызывается всего один раз и ее вовсе не следует делать подпрограммой. В результате анализа может оказаться, что некая подпрограмма никогда не вызывается и ее можно удалить.

На этом этапе рекомендуется тщательно все проверить и снова начать с этапа 1 или даже с этапа 0! Это никакая не шутка, поскольку время, затраченное на эти действия, не будет потрачено зря, ведь таким образом можно досконально изучить данную проблему. Оглядываясь на пройденный путь, вам, вероятно, удастся найти несколько улучшений в общей организации программы. Нет никаких оснований для опасений при возврате к этапу 1, и еще меньше их будет при повторном проходе от этапа 2 к этапу 3, ведь с подобной программой уже приходилось иметь дело. Более того, вполне вероятно эта работа позднее позволит сэкономить время отладки, когда снова придется переписывать код программы. Некоторые наиболее удачные компьютерные программы стали успешными только благодаря случайной утрате всего кода приблизительно на этом этапе работы, и их авторам приходилось начинать все сначала.

С другой стороны, вероятно, не существует такой сложной компьютерной программы, которую нельзя было бы улучшить, а потому не следует до бесконечности повторять этапы 1 и 2. После начальных значительных улучшений, в конечном итоге, всегда наступает момент, когда получаемая выгода не оправдывает затраченного времени.

Этап 4 (Отладка). После окончательной полировки программы, которая вероятно включает выделение памяти и другие детали, пора обратить внимание на другие вопросы, которые не учитывались на этапах 1, 2 и 3. Теперь нужно проанализировать программу в том порядке, в котором она *выполняется* компьютером. Это можно сделать вручную или, конечно же, с помощью компьютера. Автор считает весьма полезным на этом этапе использовать системные процедуры для трассировки каждой инструкции во время нескольких первых попыток их выполнения. Здесь важно еще раз представить идеи, лежащие в основе программы и проверить как они соответствуют выполняемым действиям.

Отладка — это искусство, которое требует отдельного изучения. Сам процесс отладки очень сильно зависит от имеющихся на данном компьютере соответствующих инструментов. Для эффективной отладки рекомендуется использовать подходящие проверочные данные. Наиболее успешные методы отладки обычно продумываются и встраиваются в саму программу. Многие наиболее успешные современные программисты посвящают почти половину своей программы процедурам отладки другой половины программы. Первая часть программы, которая обычно состоит из очень простых частей, отображающих информацию в читабельном формате, казалось бы не имеет большого значения, но в общем итоге это дает значительное увеличение производительности.

Еще один эффективный метод отладки заключается в том, чтобы учитывать все замеченные ошибки. Даже если эта информация порой выглядит очень неприглядно, она имеет бесценное значение для отладки программы и, несомненно, поможет справиться с подобными ошибками в будущем.

Примечание: автор впервые сформулировал эти соображения в 1964 году, после успешного завершения нескольких проектов среднего размера по созданию программного обеспечения, но еще до создания зрелого стиля программирования. Позднее, в 1980-х годах, автор узнал о существовании другой технологии, *структурного документирования*, или *грамотного программирования*, которая, вероятно, имеет большее значение. Итоговая сводка современных представлений автора о наилучших способах создания программ разных видов представлена в его книге *Literate Programming* (Cambridge University Press, впервые опубликована в 1992). Совершенно случайно в главе 11 этой книги приводится подробный перечень всех ошибок, которые удалось найти и устранить в программе TEX в 1978–1991 годы.

Уж лучше программировать с ошибками (до некоторой степени),
чем тратить столько времени на проектирование безошибочных программ
(сколько еще десятилетий потребуется для этого?).

— А. М. ТЬЮРИНГ (A. M. TURING), Предложения для ACE (Proposals for ACE) (1945)

УПРАЖНЕНИЯ:

1. [*20*] Создайте подпрограмму `GoMaxR`, которая обобщала бы алгоритм 1.2.10M поиска максимального значения среди $\{X[a], X[a+r], X[a+2r], \ldots, X[n]\}$, где r и n являются положительными параметрами и a является наименьшим положительным числом $a \equiv n$ (по модулю r), а именно $a = 1 + (n-1) \bmod r$. Создайте специальный вход `GoMax` для случая $r = 1$, с помощью вызывающей последовательности на основе `GO`, чтобы полученная подпрограмма была обобщением (7).

2. [*20*] В подпрограмме из упр. 1 замените связывание на основе инструкции `GO` на другой тип связывания на основе инструкций `PUSHJ/POP`.

3. [*15*] Как можно было бы упростить схему (15), если известно, что `Sub` является концевой подпрограммой?

4. [*15*] В этом разделе в основном говорится о `PUSHJ`, а в разделе 1.3.1′ упоминается только команда `PUSHGO`. Чем отличаются `PUSHJ` и `PUSHGO`?

5. [*0*] Истинно или ложно следующее утверждение: количество маргинальных регистров равно $G - L$.

6. [*10*] Каким будет результат выполнения инструкции `DIVU $5,$5,$5`, если $5 является маргинальным регистром?

7. [*10*] Каким будет результат выполнения инструкции `INCML $5,#abcd`, если $5 является маргинальным регистром?

8. [*15*] Предположим, что инструкция `SET $15,0` выполняется при наличии 10 локальных регистров. Это приводит к увеличению числа локальных регистров до 16, но новые локальные регистры (включая $15) равны нулю, потому они ведут себя так, как если бы были маргинальными. Является ли инструкция `SET $15,0` избыточной в этом случае?

9. [*28*] Если прерывание обхода используется для такой исключительной ситуации, как арифметическое переполнение, то обработчик обхода может вызываться в непредсказуемые моменты времени. В таких случаях нежелательно засорять регистры прерванной программы. Обработчик обхода не может сильно навредить, если только не давать ему "где развернуться". Объясните, как можно было бы использовать инструкции `PUSHJ` и `POP`, чтобы достаточное количество локальных регистров было доступно для обработчика.

▸ **10.** [*20*] Истинно или ложно следующее утверждение: если программа `MMIX` никогда не использует инструкции `PUSHJ`, `PUSHGO`, `POP`, `SAVE` или `UNSAVE`, то все 256 регистров \$0, \$1, ..., \$255 эквивалентны в том смысле, что нет различий между локальными, глобальными и маргинальными регистрами.

11. [*20*] Что случиться, если программа попытается выполнить больше инструкций `POP`, чем инструкций `PUSH`.

▸ **12.** [*10*] Истинно или ложно следующее утверждение:
a) Текущий префикс в программе `MMIXAL` всегда начинается с двоеточия.
b) Текущий префикс в программе `MMIXAL` всегда заканчивается двоеточием.
c) Символы `:` и `::` эквивалентны в программе `MMIXAL`.

▸ **13.** [*21*] Создайте две подпрограммы `MMIX` для вычисления чисел Фибоначчи $F_n \bmod 2^{64}$ для заданного n.

Первая подпрограмма должна рекурсивно вызывать сама себя, согласно определению:

$$F_n = n \quad \text{если } n \le 1; \qquad F_n = F_{n-1} + F_{n-2} \quad \text{если } n > 1.$$

Вторая подпрограмма *не* должна быть рекурсивной. Обе подпрограммы должны использовать связывание на основе инструкций `PUSH/POP` и не использовать глобальных переменных.

▸ **14.** [*M21*] Каково время выполнения подпрограмм в упражнении 13?

▸ **15.** [*21*] Преобразуйте рекурсивную подпрограмму из упражнения 13 для создания подпрограммы со связыванием на основе инструкций `GO`, используя стековую память, как в (15) вместо стека регистров `MMIX`. Сравните эффективность двух версий.

▸ **16.** [*25*] (*Нелокальный оператор* ***goto***.) Иногда требуется перейти из подпрограммы в место, расположенное вне вызывающей процедуры. Предположим, что подпрограмма `A` вызывает подпрограмму `B`, которая вызывает подпрограмму `C`, которая рекурсивно вызывает сама себя несколько раз перед тем, как решить выйти непосредственно в подпрограмму `A`. Объясните, как можно было бы обработать такую ситуацию с помощью стека регистров `MMIX`(Нельзя просто так перейти с помощью инструкции `JMP` из `C` в `A`; стек должен быть обработан соответствующим образом.)

1.4.2′. Сопрограммы

Подпрограммы являются особыми случаями более общих программных компонентов, *сопрограмм*. В отличие от несимметричной связи между основной программой и подпрограммой, между сопрограммами имеется полная симметрия, которая выражается в том, что они могут *вызывать друг друга*.

Для понимания концепции сопрограммы, рассмотрим следующую ситуацию. В предыдущем разделе предлагалась точка зрения, что подпрограмма является всего лишь своеобразным расширением аппаратного обеспечения компьютера, предназначенного для экономии размера кода программы. Это верно, но существует и другая точка зрения: основную программу и подпрограмму можно считать *командой* программ, где каждый член команды имеет определенный участок работы. Основная программа на своем участке работы, активизирует подпрограмму, а подпрограмма выполняет свои функции и активизирует основную программу. Расширим рамки своего воображения и представим эту ситуацию с точки зрения подпрограммы: при выходе подпрограммы *она* вызывает *основную* программу, а основная программа продолжает свою работу и затем “выходит” в подпрограмму.

В свою очередь, подпрограмма по окончании своей работы снова вызывает основную программу.

Такой равноправный философский подход может показаться притянутым за уши, но он справедлив по отношению к сопрограммам. Дело в том, что при работе с сопрограммами нельзя сказать, какая из них подчиняется другой. Допустим, что программа состоит из сопрограмм A и B, тогда при программировании A сопрограмма B считается подпрограммой, а при программировании B сопрограмма A считается подпрограммой. Всякий раз при активизации "подчиненной' сопрограммы, выполнение "основной' сопрограммы продолжается с места ее приостановки.

Сопрограммы A и B могут, например, быть двумя программами-игроками в шахматы, т.е. их можно скомбинировать так, чтобы они играли в шахматы друг с другом.

Такое связывание сопрограмм легко осуществляется в MMIX, если применить два глобальных регистра, a и b. В сопрограмме A инструкция "GO a,b,0" используется для активизации сопрограммы B, а в сопрограмме B инструкция "GO b,a,0" используется для активации сопрограммы A. В этой схеме для передачи управления в любом направлении требуется всего $3v$ единиц времени.

Существенное отличие между связыванием программы-подпрограммы и сопрограммы-сопрограммы можно обнаружить, сравнивая связывание на основе GO из предыдущего раздела со следующей схемой. Подпрограмма всегда запускается *с ее начала*, которое располагается в заданном месте, а основная программа или сопрограмма всегда запускается *с места после* ее последнего прерывания.

Сопрограммы наиболее естественным образом возникают на практике при работе с алгоритмами ввода и вывода. Например, допустим, что сопрограмма A должна считать файл и выполнить некоторые преобразования входных данных, сводя их к последовательности элементов. Другая сопрограмма, назовем ее B, выполняет дальнейшую обработку этих элементов и выводит полученный результат. Сопрограмма B периодически вызывает последовательные элементы входных данных с помощью A. Таким образом, сопрограмма B переходит к сопрограмме A всякий раз, когда нужен следующий элемент входных данных, а сопрограмма A переходит к сопрограмме B всякий раз, когда найден следующий элемент входных данных. Читатель может заметить: "Хорошо, B является основной программой и A является всего лишь *подпрограммой* для предоставления входных данных". Это замечание, однако, становится менее справедливым, если процесс A заметно усложнится. Действительно, наоборот можно представить, что A является основной программой, B является подпрограммой для предоставления выходных данных, а приведенное выше описание остается справедливым. Польза от введения понятия "сопрограмма" проявляется на стыке между этими двумя крайними ситуациями, когда A и B имеют сложную структуру и каждая из них многократно вызывает другую. Не легко найти короткий и простой пример использования сопрограмм, которые иллюстрировали бы значение этой идеи, поскольку наиболее полезные сопрограммы обычно имеют довольно большой размер.

Для более тщательного изучения сопрограмм рассмотрим следующий притянутый за уши пример. Предположим, что нужно создать программу, которая транслирует один код в другой. Входной код, который нужно транслировать в последо-

вательность 8-битовых символов, заканчивается точкой, как показано ниже

```
a2b5e3426fg0zyw3210pq89r.
```
(1)

Этот код находится в стандартном входном файле, который перемежается "пробельными символами" произвольного вида. В данном случае "пробельным символом" будем считать любой байт, значение которого меньше или равно #20, которое является символьной константой ' ' в программах MMIXAL. При считывании входных данных все символьные пробелы игнорируются, а другие символы интерпретируются следующим образом: (1) если следующий символ является одной из десятичных цифр 0 или 1 или $\cdots$ или 9, например n, то нужно $(n+1)$ раз повторить следующий символ, не важно, является он цифрой или нет. (2) нецифровой символ просто воспроизводится. Выход программы должен состоять из последовательности разделенной на группы по три символа до тех пор, пока не появится точка. Последняя группа может иметь меньше трех символов. Например, (1) транслируется в

```
abb bee eee e44 446 66f gzy w22 220 0pq 999 999 999 r.
```
(2)

Обратите внимание, что код 3426f означает не 3427 букв f, а 4 четверки, 3 шестерки и одну букву f. Если входной код имеет вид '1.', то на выходе будет '.', а не '..', поскольку точка прекращает вывод. Цель нашей программы состоит в создании последовательности строк в стандартном выходном файле с 16 трехсимвольными группами в строке (за исключением, конечно, последней строки, которая может быть короче). Трехсимвольные группы нужно разделять пробелами, а каждую строку завершать символом ASCII новой строки #a.

Для выполнения этой трансляции создадим две сопрограммы и одну подпрограмму. Программа начинается с присвоения символьных имен трем глобальным регистрам: одно для временного хранения и остальные для связывания сопрограмм.

```
01  * Пример сопрограмм.
02  t       IS    $255  Временные данные кратковременного хранения.
03  in      GREG  0     Адрес для продолжения первой сопрограммы.
04  out     GREG  0     Адрес для продолжения второй сопрограммы.  ▮
```

Следующий шаг заключается в определении участков памяти для хранения рабочих данных.

```
05  * Входной и выходной буферы.
06            LOC   Data_Segment
07            GREG  @              Базовый адрес.
08  OutBuf    TETRA "                ",#a,0  (см. упражнение 3)
09  Period    BYTE  '.'
10  InArgs    OCTA  InBuf,1000
11  InBuf     LOC   #100             ▮
```

Теперь вернемся к самой программе. Подпрограмма NextChar предназначена для поиска непробельных символов входа и возвращения следующего такого символа:

```
12  * Подпрограмма символьного входа
13  inptr     GREG  0              Текущая входная позиция.
14  1H        LDA   t,InArgs       Заполнить входной буфер.
```

```
15           TRAP  0,Fgets,StdIn
16           LDA   inptr,InBuf     Стартовать в начале буфера.
17  0H       GREG  Period
18           CSN   inptr,t,0B      Если произошла ошибка, считать '.'.
19  NextChar LDBU  $0,inptr,0      Извлечь следующий символ.
20           INCL  inptr,1
21           BZ    $0,1B           Условный переход, если конец буфера.
22           CMPU  t,$0,' '
23           BNP   t,NextChar      Условный переход, если пробельный символ.
24           POP   1,0             Вернуться в вызывающую программу. ▌
```

Эта подпрограмма обладает следующими характеристиками:

Вызывающая последовательность: `PUSHJ $R,NextChar`.
Условия входа: `inptr` указывает на первый несчитанный символ.
Условия выхода: $R = следующий непробельный символ; `inptr` готов для следующего элемента `NextChar`.

Подпрограмма также изменяет регистр `t`, а именно регистр $255, но мы обычно опускаем этот регистр в таких спецификациях, как это было в 1.4.1′–(10).

Наша первая сопрограмма, `In`, находит символы во входном коде и определяет количество повторов. Она начинает работу с позиции `In1`:

```
25  * Первая сопрограмма
26  count    GREG  0               Счетчик повторов.
27  1H       GO    in,out,0        Послать символ сопрограмме Out.
28  In1      PUSHJ $0,NextChar     Получить новый символ.
29           CMPU  t,$0,'9'
30           PBP   t,1B            Условный переход, если превышает '9'.
31           SUB   count,$0,'0'
32           BN    count,1B        Условный переход, если меньше '0'.
33           PUSHJ $0,NextChar     Получить новый символ.
34  1H       GO    in,out,0        Передать его в Out.
35           SUB   count,count,1   Уменьшить счетчик повторов.
36           PBNN  count,1B        Повторить, если необходимо.
37           JMP   In1             В противном случае начать новый цикл. ▌
```

Эта сопрограмма обладает следующими характеристиками.

Вызывающая последовательность (из `Out`): `GO out,in,0`.
Условия выхода (в `Out`): $0 = следующий входной символ с определенным числом повторов.
Условия входа (после возвращения): $0 не отличается от значения на выходе.

Регистр `count` является частным для `In` и нуждается в упоминании.

Другая сопрограмма, `Out`, размещает код в трехсимвольных группах и посылает их в стандартный выходной файл. Она начинает работу с позиции `Out1`:

```
38  * Вторая сопрограмма
39  outptr   GREG  0               Текущая позиция выхода.
40  1H       LDA   t,OutBuf        Опустошить выходной буфер.
41           TRAP  0,Fputs,StdOut
42  Out1     LDA   outptr,OutBuf   Стартовать с начала буфера.
43  2H       GO    out,in,0        Получить новый символ из In.
44           STBU  $0,outptr,0     Сохранить его как первый из трех.
45           CMP   t,$0,'.'
46           BZ    t,1F            Условный переход, если '.'.
47           GO    out,in,0        В противном случае получить другой символ.
48           STBU  $0,outptr,1     Сохранить его как второй из трех.
49           CMP   t,$0,'.'
50           BZ    t,2F            Условный переход, если '.'.
51           GO    out,in,0        В противном случае получить другой символ.
52           STBU  $0,outptr,2     Сохранить его как третий из трех.
53           CMP   t,$0,'.'
54           BZ    t,3F            Условный переход, если '.'.
55           INCL  outptr,4        В противном случае перейти к следующей
                                   группе.
56  0H       GREG  OutBuf+4*16
57           CMP   t,outptr,0B
58           PBNZ  t,2B            Условный переход, если меньше 16 групп.
59           JMP   1B              В противном случае завершить строку.
60  3H       INCL  outptr,1        Пройти мимо сохраненного символа.
61  2H       INCL  outptr,1        Пройти мимо сохраненного символа.
62  0H       GREG  #a              (символ новой строки)
63  1H       STBU  0B,outptr,1     Сохранить символ новой строки после точки.
64  0H       GREG  0               (пустой символ)
65           STBU  0B,outptr,2     Сохранить пустой символ после новой строки.
66           LDA   t,OutBuf
67           TRAP  0,Fputs,StdOut  Вывести последнюю строку.
68           TRAP  0,Halt,0        Завершить программу. ▮
```

Характеристики `Out` задуманы такими, чтобы они дополняли характеристики `In`.

Вызывающая последовательность (из `In`):	`GO in,out,0`.
Условия выхода (в `In`):	$0 не отличается от значения на входе.
Условия входа (upon return):	$0 = следующий входной символ с определенным числом повторов.

Для завершения программы нужно задать условия запуска. Инициализация сопрограммы имеет замысловатый вид, хотя, на самом деле, она не так сложна, как кажется.

```
69  * Инициализация
70  Main     LDA   inptr,InBuf   Инициализировать NextChar.
71           GETA  in,In1        Инициализировать In.
72           JMP   Out1          Стартовать с Out (см. упражнение 2). ▮
```

Работа над программой завершена. Читателю рекомендуется тщательно изучить ее, обращая особое внимание на то, как каждая сопрограмма может считы-

ваться и создаваться независимо, как если бы другая сопрограмма была ее подпрограммой.

Из раздела 1.4.1′ известно, что инструкции `PUSHJ` и `POP` компьютера `MMIX` предпочтительнее, чем команда `GO`, при организации связывания подпрограмм. Но для сопрограмм верно противоположное утверждение: инструкции `PUSHJ` и `POP` очень несимметричны, а стек регистра компьютера `MMIX` может быть полностью запутан, если две или более сопрограмм попытаются одновременно использовать его. (См. упражнение 6.)

Между сопрограммами и *многопроходными алгоритмами* есть важная связь. Например, описанный выше процесс трансляции можно выполнить за два прохода. Сначала можно было бы выполнить сопрограмму `In`, применяя ее ко всему входу и записывая каждый символ с определенным количеством повторов в промежуточный файл. После этого можно было бы считать файл и выполнить сопрограмму `Out`, принимая символы в группах по три элемента. В этом состоит суть "двухпроходного" процесса. (Интуитивно понятно, что "проход" обозначает полное сканирование входа. Это определение не совсем точно и во многих алгоритмах не ясно, какое количество проходов используется, но интуитивная концепция "прохода" все же полезна, несмотря на ее нечеткое определение.)

На рис. 22а показан четырехпроходный процесс. Довольно часто можно обнаружить, что тот же процесс можно выполнить всего за один проход, как показано на рис. 22б, если подставить четыре сопрограммы `A`, `B`, `C`, `D` для соответствующих проходов `A`, `B`, `C`, `D`. Сопрограмма `A` перейдет к сопрограмме `B`, вместо записи результата прохода `A` в файл 1. Сопрограмма `B` перейдет к сопрограмме `A`, вместо чтения входа из Файла 1, потом `B` перейдет к сопрограмме `C`, вместо записи результата прохода в Файл 2; и т.д. Пользователям UNIX® эта конструкция известна под названием "канал" или "конвейер" и обозначается "`Проход A | Проход B | Проход C | Проход D`". Программы для проходов `B`, `C` и `D` иногда называются "фильтрами".

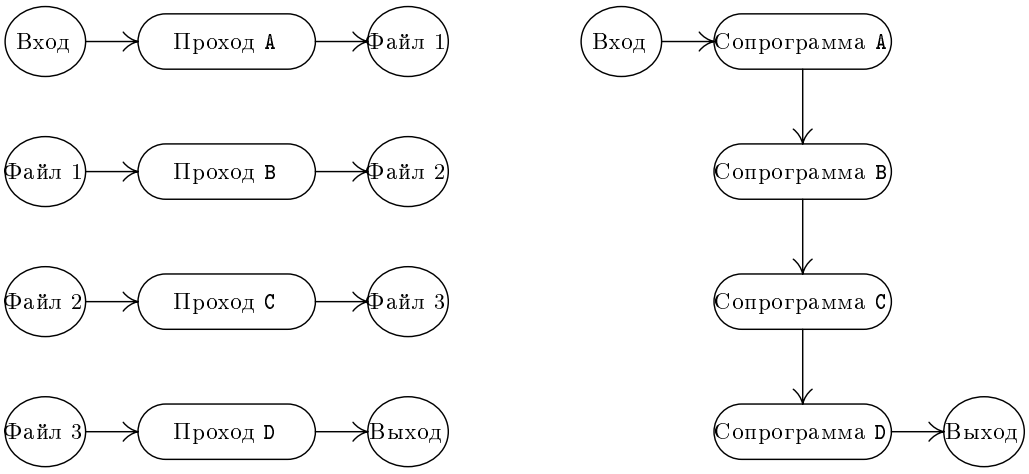


Рис. 22. Проходы: а — четырехпроходной алгоритм; б — однопроходной алгоритм

И наоборот, процесс, выполняемый n сопрограммами, часто трансформируется в n-проходный процесс. В связи с этим, имеет смысл сравнить многопроходные и однопроходные алгоритмы.

a) *Психологическое отличие.* Многопроходный алгоритм обычно проще создать и понять, чем однопроходный алгоритм для той же проблемы. Процесс, разделенный на несколько малых этапов, которые идут друг за другом, легче для понимания, чем единый сложный процесс, в котором одновременно происходит несколько преобразований.

Кроме того, если имеется крупная задача и для создания программы для ее решения предполагается коллективная работа многих людей, то многопроходной алгоритм представляет собой естественный способ разделения участков работы между ними.

Преимущества многопроходного алгоритма выражаются также в сопрограммах, поскольку каждая сопрограмма может создаваться отдельно от других сопрограмм. Их связывание превращает кажущийся многопроходный алгоритм в однопроходный процесс.

b) *Временное отличие.* Время, необходимое на упаковку, создание, чтение и распаковку промежуточных данных между потоками (например, в виде файлов на рис. 22), исключается в однопроходном алгоритме. По этой причине, однопроходный алгоритм будет выполняться быстрее.

c) *Пространственное отличие.* При использовании однопроходного алгоритма требуется пространство в памяти для одновременного хранения всех программ, а при использовании многопроходного алгоритма требуется пространство в памяти для одновременного хранения только одной сопрограммы. Эта особенность может повлиять на скорость выполнения программы еще в большей степени, чем указано в пункте (b). Например, многие компьютеры обладают ограниченным объемом "быстрой памяти" и большим объемом более медленной памяти. Если каждый проход едва помещается в быстрой памяти, то время выполнения многопроходного алгоритма будет гораздо меньше. (Дело в том, что использование сопрограмм может привести к тому, что большая часть программы окажется в медленной памяти или будет часто сбрасываться в/из быстрой памяти).

Иногда возникает необходимость спроектировать разные алгоритмы для нескольких компьютерных конфигураций с разным объемом памяти. В таких случаях можно написать программу в виде нескольких сопрограмм и, в зависимости от размера памяти, управлять количеством проходов: загружать приемлемое количество сопрограмм, а для организации отсутствующих связей применить подпрограммы ввода или вывода.

Хотя взаимосвязь между сопрограммами и проходами имеет большое значение, следует иметь в виду, что для приложений с сопрограммами не всегда можно реализовать многопроходной алгоритм. Если сопрограмма `B` получает входные данные от сопрограммы `A` и отправляет назад важную информацию сопрограмме `A`, как в примере с шахматами, то эту последовательность действий нельзя представить в виде прохода `A`, за которым следует проход `B`.

Наоборот, ясно, что некоторые многопроходные алгоритмы нельзя преобразовать в сопрограммы. Некоторые алгоритмы по своей природе являются многопро-

ходными. Например, для второго прохода может потребоваться такая обобщенная информация из первого прохода, как общее количество появления определенного слова во входном потоке. Вот старый анекдот, который стоит упомянуть в этом контексте.

> *Старушка в автобусе:* "Малыш, на какой остановке нужно выходить, чтобы попасть на улицу Пасадена-Стрит?"
> *Малыш:* "Просто следите за мной и выходите на две остановки раньше той, на которой выйду я."

(Суть анекдота в том, что малыш предлагает двухпроходной алгоритм для решения данной проблемы.)

Вот и все о многопроходных алгоритмах. Сопрограммы также играют важную роль в симуляции дискретных систем (см. раздел 2.2.5). Если несколько более или менее независимых сопрограмм управляются основным процессом, то их часто называют *потоками* вычислений. Другие примеры сопрограмм часто рассматриваются в других частях данной книги. В главе 8 рассматривается очень важная идея *реплицируемых сопрограмм* , а некоторые интересные приложения этой идеи представлены в главе 10.

УПРАЖНЕНИЯ:

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

▶ **2.** [*20*] Программа в этом разделе начинается с сопрограммы `Out`. Что произойдет, если она начнется с сопрограммы `In`, т.е., если строки 71 и 72 будут иметь вид "`GETA out,Out1; JMP In1`"?

3. [*15*] Объясните предназначение инструкции `TETRA` в строке 08 программы в этом разделе. (Обратите внимание, что между кавычками находится 15 пробелов.)

4. [*20*] Допустим, что две сопрограммы `A` и `B` рассматривают регистр остатка rR компьютера `MMIX` так, как если бы он был бы их закрытым элементом, хотя обе они вовлечены в процесс деления. (Иначе говоря, если одна сопрограмма передает управление другой, то содержимое rR не должно меняться, если управление возвращается другой сопрограмме.) Придумайте такое связывание сопрограмм, при котором соблюдалось бы это условие.

5. [*20*] Можно ли для `MMIX` создать достаточно эффективное связывание сопрограмм с помощью инструкций `PUSH` и `POP` без использования команд `GO`?

6. [*20*] В программе для `MMIX` в данном разделе стек регистров используется только очень ограниченным способом, а именно, когда `In` вызывает `NextChar`. В какой степени две взаимодействующие сопрограммы могут совместно использовать стек регистров?

▶ **7.** [*30*] Создайте программу `MMIX`, которая *обращает* трансляцию, выполненную программой в данном разделе. Иначе говоря, эта программа должна преобразовывать файл с трехсимвольными группами (2) в файл с кодом (1). Выход должен иметь вид максимально короткой строки символов, например, нуль перед `z` в (1) не должен входить в (2).

1.4.3′. Программы-интерпретаторы

В этом разделе рассматриваются довольно распространенные программы, которые часто называются *программами-интерпретаторами* или, короче, *интерпретаторами*. Интерпретатор — это компьютерная программа, которая выполняет инструкции другой программы, созданной на машинном языке. Под машинным

языком подразумевается способ представления инструкций на основе кодов операций (опкодов), адресов и т.д. (Это определение, как и большинство определений современных компьютерных терминов, неточно, но оно и не должно быть таким. Нельзя точно провести линию и сказать, что какие-то программы являются интерпретаторами, а какие-то — нет.)

Исторически первые интерпретаторы создавались для машинных языков, предназначенных специально для упрощения программирования. Такой интерпретируемый язык проще в употреблении, чем машинный язык. Вскоре появление символьных языков программирования исключило необходимость использования интерпретаторов такого рода, но сами интерпретаторы не исчезли. Наоборот, их использование продолжает расти, причем в такой степени, что эффективное использование интерпретаторов может рассматриваться, как одна из существенных характеристик современного программирования. Новые области приложения интерпретаторов образовались, в основном по следующим причинам:

a) машинный язык способен представить сложную последовательность решений и действий в компактной и эффективной форме;

b) такое представление предоставляет прекрасный способ сообщения между проходами в многопроходном процессе.

В таких случаях создаются специальные языки, похожие на машинные, которые используются для отдельной программы, а программы на этих языках часто генерируются только компьютерами. (Современные профессиональные программисты являются прекрасными дизайнерами компьютеров: они не только создают интерпретаторы, но и определяют *виртуальный компьютер*, язык которого нужно интерпретировать.)

Технология интерпретирования имеет еще одно дополнительное преимущество относительной независимости от компьютера. Дело в том, что только интерпретатор нужно изменить при переходе на другой тип компьютера. Более того, полезные средства отладки можно дополнительно встроить в интерпретируемые системы.

Примеры интерпретаторов типа (a) представлены в нескольких местах далее в этой серии книг. Например, рекурсивный интерпретатор в главе 8 и синтаксический анализатор (“Parsing Machine”) в главе 10. Обычно возникают ситуации, когда имеется большое количество особых случаев, все похожие, но не имеющие простой структуры.

Например, рассмотрим задачу создания алгебраического компилятора, в котором нужно эффективно генерировать инструкции машинного языка. Они складывают две величины, которые принадлежат одному из 10 классов (константы, простые переменные, индексированные переменные, переменные с фиксированной или плавающей точкой, знаковые или беззнаковые переменные и т.д.). Комбинации всех этих пар образуют 100 разных случаев. Для соответствующей обработки всех этих случаев потребуется создать очень сложную программу. Интерпретируемое решение этой задачи заключается в создании специального языка, чьи “инструкции” помещались бы в одном байте. Затем нужно подготовить таблицу 100 “программ” на этом языке, причем каждая программа умещается в одном слове. Идея заключается в том, чтобы выбрать соответствующий элемент таблицы и выполнить найденную там программу. Эта технология очень проста и эффективна.

Пример интерпретатора типа (b) представлен в статье "Computer-Drawn Flowcharts", D. E. Knuth, *CACM* **6** (1963), 555–563. В многопроходной программе ранние проходы должны передавать информацию более поздним проходам. Эта информация часто более эффективно передается с помощью машинного языка, как набор инструкций для более позднего прохода. Более поздний проход часто представляет собой не более, чем интерпретатор особого назначения, а более ранний интерпретатор представляет собой "компилятор" особого назначения. Философия многопроходной операции может быть охарактеризована, как *пересказ* более позднему проходу последовательности действий при любой возможности, а не просто представление множества фактов и просьбу *сообразить* что сделать.

Другим примером интерпретатора типа (b) являются компиляторы особых языков программирования. Если язык включает много компонентов, которые не так легко выполнить на компьютере без использования подпрограмм, то результирующая объектная программа будет содержать длительную последовательность вызовов подпрограмм. Это может произойти, например, если язык имеет дело преимущественно с высокоточными арифметическими операциями. В таком случае объектная программа будет значительно короче, если выражается на интерпретируемом языке. Например, в книге *ALGOL 60 Implementation*, by B. Randell and L. J. Russell (New York: Academic Press, 1964) описывается компилятор для трансляции ALGOL 60 в интерпретируемый язык, а также описывается интерпретатор этого языка. В книге "An ALGOL 60 Compiler," by Arthur Evans, Jr., *Ann. Rev. Auto. Programming* **4** (1964), 87–124, приводятся примеры интерпретаторов, используемых *в* компиляторе. Появление микропрограммируемых компьютеров и интегральных микросхем специального назначения еще более повысило значение интерпретируемых подходов.

Программа TEX, с помощью которой создана эта книга, преобразует текст данного раздела в интерпретируемый язык, т.е. формат `DVI`, созданный Д. Р. Фуксом в 1979. [См. D. E. Knuth, *TEX: The Program* (Reading, Mass.: Addison–Wesley, 1986), Part 31.] Полученный файл формата `DVI`, который создан с помощью TEX, потом обрабатывается интерпретатором `dvips`, созданный Т. Г. Рокицки, и преобразуется в файл инструкций на другом интерпретируемом языке PostScript® [Adobe Systems Inc., *PostScript Language Reference*, 3rd edition (Reading, Mass.: Addison–Wesley, 1999)]. Файл формата PostScript передается в издательство для печати на принтере, который использует интерпретатор PostScript для создания печатных форм. Этот трехпроходной пример иллюстрирует интерпретаторы типа (б); TEX также включает малый интерпретатор типа (а) для обработки так называемой лигатуры и кернинга печатаемых символов [*TEX: The Program*, §545].

Программу на интерпретируемом языке можно представить следующим образом: ее можно рассматривать как набор вызовов подпрограмм, один вызов за другим. Такую программу можно расширить фактически в виде длинной последовательности вызовов подпрограмм. И наоборот, такую последовательность можно упаковать в кодированной и готовой для интерпретирования форме. Преимуществами технологии интерпретирования являются компактность представления, независимость от типа компьютеров и улучшенные возможности диагностики. Интерпретатор часто можно создать таким образом, чтобы время, потраченное на

интерпретацию кода и ветвление к соответствующей процедуре, было достаточно малым.

***Симулятор MMIX.** Если язык интерпретируемой процедуры является машинным языком другого компьютера, то интерпретатор часто называется *симулятором* (или иногда *эмулятором*).

По мнению автора, чересчур много времени программистов было потрачено зря на создание таких симуляторов и чересчур много компьютерного времени было потрачено зря на их использование. Мотивация такой деятельности очень проста: при покупке нового компьютера пользователь хотел бы использовать программы, созданные для прежнего устаревшего компьютера (а не переписывать программы). Однако это обычно дороже и менее эффективно, чем перепрограммирование нужных программ с помощью временной команды программистов. Например, автору ранее приходилось участвовать в подобном проекте перепрограммирования и в ходе работы была обнаружена серьезная ошибка в исходной программе, которая использовалась в течение нескольких лет. Новая программа работала в пять раз быстрее прежней и давала правильные результаты! (Не все симуляторы плохи. Например, производители компьютеров обычно стремятся симулировать поведение нового компьютера до его постройки, чтобы как можно быстрее создать программное обеспечение для него. Однако это очень специализированный способ применения симуляторов.) Вот реальный экстремальный пример неэффективного использования компьютерного симулятора: компьютер A симулирует компьютер B, который выполняет программу, симулирующую поведение компьютера C. В этом случае большой и дорогой компьютер будет менее эффективен, чем его более дешевый собрат.

В свете сказанного выше, почему симулятору уделяется внимание в данной книге? Для этого есть следующие три причины.

a) Описываемый ниже симулятор является прекрасным примером типичной интерпретируемой процедуры. На его примере иллюстрируются базовые технологии создания интерпретаторов и применение подпрограмм в сравнительно длинной программе.

b) Здесь описывается симулятор компьютера `MMIX`, созданный (представьте себе) на языке `MMIX`. Это улучшит наши знания о данном компьютере и поможет в создании симуляторов `MMIX` для других компьютеров, хотя мы не будем погружаться в подробности 64-битовой целочисленной арифметики или вычислений с плавающей точкой.

c) Симуляция `MMIX` объясняет способы эффективной организации стека регистров на аппаратном уровне так, чтобы проталкивание и выталкивание достигалось с наименьшими затратами. Аналогично, представленный здесь симулятор проясняет поведение операторов `SAVE` и `UNSAVE`, а также дает более подробное представление о прерываниях обхода. Такие конструкции становятся более понятными при изучении способов реализации ссылок и работы компьютера.

Описываемые здесь компьютерные симуляторы следует отличать от *дискретных системных симуляторов*. Дискретные системные симуляторы имеют большое значение и рассматриваются в разделе 2.2.5.

Вернемся к задаче создания симулятора MMIX. Начнем со значительного упрощения: вместо симуляции всех одновременно происходящих процессов в конвейерном компьютере мы будем интерпретировать одновременно только по одной инструкции. Конвейерная обработка имеет чрезвычайно большое значение, но ее изучение выходит за рамки этой книги. Заинтересованные читатели могут найти полный код программы для конвейерного "мета-симулятора" в документации *MMIXware*. Здесь мы ограничимся симулятором, которому не известно о кэш-памяти, трансляции виртуальных адресов, динамическом планировании инструкций, буферах восстановления последовательности и многом другом. Более того, здесь симулируются только те инструкции, которые используются в обычных пользовательских программах MMIX. Такие привилегированные инструкции, как LDVTS, которые зарезервированы для операционной системы, будут рассматриваться, как ошибочные, если они появятся. Прерывания обхода не будут симулироваться нашей программой, если только они не выполняют рудиментарные операции ввода или вывода, как описано в разделе 1.3.2′.

Входом нашей программы будет двоичный файл, в котором указано исходное состояние памяти, как если бы память была задана операционной системой при выполнении пользовательской программы (включая данные командной строки). Попробуем имитировать поведение аппаратной части MMIX, считая, что MMIX сам интерпретирует инструкции, которые начинаются с символьного обозначения Main. Таким образом, попробуем реализовать спецификации, представленные в разделе 1.3.1′, в среде выполнения, которая описывается в разделе 1.3.2′. Например, в программе используется массив из 256 октабайт g[0], g[1], ..., g[255] для симулированных глобальных регистров. Первые 32 элемента этого массива будут специальными регистрами, перечисленными в табл. 1.3.1′–2; один из этих специальных регистров будет симулировать часы; rC. Предполагается, что для выполнения каждой инструкции требуется определенное время, согласно таблице 1.3.1′–1. Симулированные часы rC увеличивают свои показания на 2^{32} для каждого μ и на 1 для каждого υ. Таким образом, например, после симулирования программы 1.3.2′P симулированные часы rC будут иметь значение #0000 3228 000b b091, которое представляет $12840\mu + 766097\upsilon$.

Программа очень длинна, но она содержит несколько интересных мест, и мы кратко рассмотрим наиболее простые фрагменты. Как обычно, она начинается с определения новых символов и указания содержимого сегмента данных. В этом сегменте сначала располагается массив из 256 симулированных глобальных регистров. Например, симулированный глобальный регистр $255 будет октабайтом g[255] по адресу Global+8*255. За этим глобальным массивом располагается аналогичный массив, который называется *кольцом локальных регистров*, где будут храниться самые верхние элементы симулированного стека регистров. Размер кольца равен 256, хотя 512 или еще большее значение в степени 2 также сгодится. (Большое кольцо локальных регистров потребует больше затрат, но оно дает выигрыш в скорости, если в программе часто используется стек регистров. Одно из предназначений симулятора состоит в определении целесообразности использования дополнительного аппаратного обеспечения.) Основная часть сегмента данных, начиная с Chunk0, посвящается симулированной памяти.

```
001  * Симулятор MMIX (упрощенный)
002  t        IS   $255              Регистр временной информации.
003  lring_size IS 256               Размер кольца локальных регистров.
004           LOC  Data_Segment      Начать с адреса #2000000000000000.
005  Global   LOC  @+8*256           256 октабайт для глобальных регистров.
006  g        GREG Global            Базовый адрес глобальных регистров.
007  Local    LOC  @+8*lring_size    lring_size октабайт для локальных регистров.
008  l        GREG Local             Базовый адрес для локальных регистров.
009           GREG @                 Базовый адрес для IOArgs и Chunk0.
010  IOArgs   OCTA 0,BinaryRead      (См. упражнение 20.)
011  Chunk0   IS   @                 Начало области симулированной памяти.
012           LOC  #100              Остальное в текстовый сегмент. ▮
```

Одна из наиболее важных подпрограмм называется `MemFind`. Имея 64-битовый адрес A, эта подпрограмма возвращает результирующий результат R, по которому находится симулированное содержание $M_8[A]$. Конечно, 2^{64} байт симулированной памяти нельзя втиснуть в 2^{61}-байтовый сегмент данных. Но симулятор запоминает все встречавшиеся прежде адреса и предполагает, что все невстречавшиеся адреса равны нулю.

Память делится на "порции" по 2^{12} байт каждая. `MemFind` анализирует первые $64 - 12 = 52$ бит A, чтобы определить их принадлежность к порции, и расширяет список известных порций в случае необходимости. Затем вычисляется R за счет сложения 12 последних бит A со стартовым адресом соответствующей симулированной порции. (Размер порции должен быть степенью 2, если каждая порция содержит по крайней мере октабайт. При работе с малыми порциями `MemFind` приходится вести поиск в длинных списках подручных порций, а при работе с большими порциями `MemFind` приходится расходовать место для байтов, к которым никогда не будет осуществляться доступ.)

Каждая симулированная порция заключена в "узел", который занимает $2^{12} + 24$ байт памяти. Первый октабайт такого узла, допустим `KEY`, идентифицирует симулированный адрес первого байта в порции. Второй октабайт, допустим `LINK`, указывает на следующий узел в списке `MemFind`, и он равен нулю для последнего узла в списке. За `LINK` следует 2^{12} байт симулированной памяти, `DATA`. Наконец, каждый узел завершается восемью нулевыми байтами, которые используются для дополнения при реализации ввода-вывода (см. упражнения 15–17).

`MemFind` поддерживает список узлов порций в порядке использования: первый узел, указанный `head`, является тем, который `MemFind` нашел в предыдущем вызове, а он связывается со следующей наиболее недавно использованной порцией и т.д. Если будущее подобно прошлому, то `MemFind` не должен искать далеко в списке. (В разделе 6.1 подробно обсуждается такая "самоорганизация" поиска в списках.) Сначала `head` указывает на порцию `Chunk0`, где `KEY`, `LINK` и `DATA` являются нулями. Указатель выделения `alloc` устанавливается в адрес, где появится узел следующей порции в случае необходимости, а именно `Chunk0+nodesize`.

`MemFind` реализуется с помощью операции `PREFIX` языка `MMIXAL`, описанной в разделе 1.4.1′, чтобы зарезервированные (приватные) символы `head`, `key`, `addr` и т.д. не конфликтовали с любыми символами в остальной части программы.

Вызывающая последовательность будет иметь вид

```
SET arg,A;  PUSHJ res,MemFind                    (1)
```

после которой результирующий адрес R появится в регистре `res`.

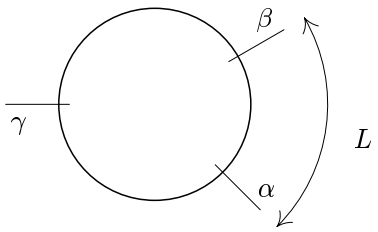
```
013           PREFIX :Mem:          (Зарезервированные символы для MemFind.)
014  head     GREG   0              Адрес первой порции.
015  curkey   GREG   0              KEY(head)
016  alloc    GREG   0              Адрес следующей выделяемой порции.
017  Chunk    IS     #1000          Байты для порции, должны быть степенью 2.
018  addr     IS     $0             Заданный адрес A.
019  key      IS     $1             Адрес порции.
020  test     IS     $2             Временный регистр для поиска ключа.
021  newlink  IS     $3             Второй наиболее недавно использованный узел.
022  p        IS     $4             Временный регистр указателя.
023  t        IS     :t             Внешний временный регистр.
024  KEY      IS     0
025  LINK     IS     8
026  DATA     IS     16
027  nodesize GREG   Chunk+3*8
028  mask     GREG   Chunk-1
029  :MemFind ANDN   key,addr,mask
030           CMPU   t,key,curkey
031           PBZ    t,4F           Ветвление, если head является правой порцией.
032           BN     addr,:Error    Не допустить отрицательные адреса A.
033           SET    newlink,head   Подготовиться для петли поиска.
034  1H       SET    p,head         p ← head
035           LDOU   head,p,LINK    head ← LINK(p)
036           PBNZ   head,2F        Ветвление, если head ≠ 0.
037           SET    head,alloc     В противном случае выделить новый узел.
038           STOU   key,head,KEY
039           ADDU   alloc,alloc,nodesize
040           JMP    3F
041  2H       LDOU   test,head,KEY
042           CMPU   t,test,key
043           BNZ    t,1B           Возврат к началу цикла, если KEY(head) ≠ key.
044  3H       LDOU   t,head,LINK    Настроить указатели: t ← LINK(head),
045           STOU   newlink,head,LINK      LINK(head) ← newlink,
046           SET    curkey,key             curkey ← key,
047           STOU   t,p,LINK               LINK(p) ← t.
048  4H       SUBU   t,addr,key     t ← отступ порции.
049           LDA    $0,head,DATA   $0 ← адрес DATA(head).
050           ADDU   $0,t,$0
051           POP    1,0            Вернуть R.
052           PREFIX :              (Конец префикса ':Mem:'.)
053  res      IS     $2             Регистр результата PUSHJ.
054  arg      IS     res+1          Регистр аргумента PUSHJ.  ▌
```

Далее рассмотрим наиболее интересный аспект симулятора, а именно: реализацию стека регистров MMIX. Как известно из раздела 1.4.1′, стек регистров

концептуально является списком τ элементов $S[0]$, $S[1]$, ..., $S[\tau-1]$. Заключительный член $S[\tau-1]$ является "вершиной" стека, а локальные регистры MMIX \$0, \$1, ..., \$$(L-1)$ являются самыми верхними L элементами $S[\tau-L]$, $S[\tau-L+1]$, ..., $S[\tau-1]$, где L является значением специального регистра rL. Стек можно было бы симулировать, просто полностью сохраняя его в симулированной памяти, но для эффективной работы компьютер должен иметь возможность мгновенного доступа к регистрам, а не к относительного медленному модулю памяти. Следовательно, будет симулироваться эффективная схема, которая хранит самые верхние элементы стека в массиве внутренних регистров, которые называются *кольцом локальных регистров*.

Основная идея чрезвычайно проста. Предположим, что кольцо локальных регистров имеет ρ элементов, l[0], l[1], ..., l[$\rho-1$]. Тогда локальный регистр \$$k$ будет храниться в l[$(\alpha+k) \bmod \rho$], где α является соответствующим отступом. (Значение ρ выбрано так, чтобы быть степенью 2, чтобы для вычисления остатка от деления mod ρ не требовалось прилагать больших усилий. Более того, ρ равно 256, по крайней мере, чтобы хватило места для всех локальных регистров.) Операция PUSH, которая перенумеровывает локальные регистры так, чтобы, например, регистр \$3 теперь назывался \$0, просто увеличивает значение α на 3. Операция POP восстанавливает предыдущее состояние за счет уменьшения α. Хотя регистры изменяют свое значение, на самом деле никакие данные не проталкиваются вниз и не выталкиваются вверх.

Конечно, потребуется использовать память для резервного копирования данных при возрастании размеров стека. Состояние кольца в любой момент времени лучше всего представить с помощью трех переменных, α, β и γ:



(2)

Элементы кольца l[α], l[$\alpha+1$], ..., l[$\beta-1$] являются текущими локальными регистрами \$0, \$1, ..., \$$(L-1)$; элементы кольца l[$\beta$], l[$\beta+1$], ..., l[$\gamma-1$] не используются в данный момент; а элементы l[γ], l[$\gamma+1$], ..., l[$\alpha-1$] содержат протолкнутые вниз элементы стека регистров. Если $\gamma \neq \alpha$, то можно увеличить γ на 1, если сначала сохранить l[γ] в памяти. Если $\gamma \neq \beta$, то можно уменьшить γ на 1, если затем загрузить l[γ]. В MMIX имеется два специальных регистра, которые называются *указателем стека* rS и *отступом стека* rO, хранящие адреса расположения l[γ] and l[α], в случае необходимости. Значения α, β и γ связаны с rL, rS и rO формулами

$$\alpha = (\mathrm{rO}/8) \bmod \rho, \quad \beta = (\alpha + \mathrm{rL}) \bmod \rho, \quad \gamma = (\mathrm{rS}/8) \bmod \rho. \tag{3}$$

Симулятор хранит большинство специальных регистров MMIX в первых 32 позициях массива глобальных регистров. Например, симулированный регистр остатка от деления rR является октабайтом, который находится по адресу `Global+8*rR`. Но восемь специальных регистров, включая rS, rO, rL и rG, потенциально относятся

к каждой симулированной инструкции, потому симулятор хранит их отдельно в собственных глобальных регистрах. Таким образом, например, регистр `ss` содержит симулированное значение rS, а регистр `ll` хранит восьмикратное симулированное значение rL:

```
055  ss  GREG  0  Симулированный указатель стека, rS.
056  oo  GREG  0  Симулированный отступ стека, rO.
057  ll  GREG  0  Симулированный локальный регистр порога, rL, умноженный на 8.
058  gg  GREG  0  Симулированный глобальный регистр порога, rG, умноженный на 8.
059  aa  GREG  0  Симулированный регистр арифметического состояния, rA.
060  ii  GREG  0  Симулированный счетчик интервала, rI.
061  uu  GREG  0  Симулированный счетчик использования, rU.
062  cc  GREG  0  Симулированный счетчик циклов, rC.  ▮
```

Вот подпрограмма, которая получает текущее значение симулированного регистра $\$k$, given k. Вызывающая последовательность имеет вид

```
SLU arg,k,3;  PUSHJ res,GetReg                    (4)
```

и искомое значение будет находиться в `res`.

```
063  lring_mask GREG  8*lring_size-1
064  :GetReg    CMPU  t,$0,gg          Подпрограмма для получения $k.
065             BN    t,1F             Условный переход, если k < G.
066             LDOU  $0,g,$0          В противном случае $k является
                                       глобальным; загрузить g[k].
067             POP   1,0              Вернуть результат.
068  1H         CMPU  t,$0,ll          t ← [$k is является локальным].
069             ADDU  $0,$0,oo
070             AND   $0,$0,lring_mask
071             LDOU  $0,l,$0          Загрузить l[(α + k) mod ρ].
072             CSNN  $0,t,0           Обнулить, если $k является
                                       маргинальным.
073             POP   1,0              Вернуть результат.  ▮
```

Обратите внимание на двоеточие в поле метки строки 064. Оно избыточно, поскольку текущим префиксом является ‘:’ (см. строку 052). Однако, двоеточие в строке 029 необходимо для внешнего символа `MemFind`, поскольку в тот момент текущим префиксом был‘`:Mem:`’. Двоеточия в поле метки, избыточны они или нет, Представляют собой удобный способ для оповещения того факта, что определяется подпрограмма.

Следующие подпрограммы, `StackStore` и `StackLoad`, симулируют операции увеличения γ на 1 и уменьшения γ на 1 в диаграмме (2). Они не возвращают результат. Подпрограмма `StackStore` вызывается, только если $\gamma \neq \alpha$, а подпрограмма `StackLoad` вызывается, только если $\gamma \neq \beta$. В обеих подпрограммах необходимо сохранять и восстанавливать регистр rJ, поскольку данные подпрограммы не являются концевыми.

```
074  :StackStore GET   $0,rJ            Сохранить адрес возврата.
075              AND   t,ss,lring_mask
076              LDOU  $1,l,t           $1 ← l[γ].
077              SET   arg,ss
```

```
078              PUSHJ res,MemFind
079              STOU  $1,res,0          M8[rS] ← $1.
080              ADDU  ss,ss,8           Увеличить rS на 8.
081              PUT   rJ,$0             Восстановить адрес возврата.
082              POP   0                 Вернуться к вызывающей программе.
083  :StackLoad  GET   $0,rJ             Сохранить адрес возврата.
084              SUBU  ss,ss,8           Уменьшить rS на 8.
085              SET   arg,ss
086              PUSHJ res,MemFind
087              LDOU  $1,res,0          $1 ← M8[rS].
088              AND   t,ss,lring_mask
089              STOU  $1,l,t            l[γ] ← $1.
090              PUT   rJ,$0             Восстановить адрес возврата.
091              POP   0                 Вернуться к вызывающей программе. ▮
```

(Регистр rJ в строках 074, 081, 083 и 090, конечно, является *реальным* rJ, а не симулированным регистром rJ. При симулировании самого компьютера нельзя забывать об этом!)

Подпрограмма `StackRoom` вызывается сразу после увеличения β. Она проверяет справедливость равенства $\beta = \gamma$ и, если оно справедливо, то увеличивает γ.

```
092  :StackRoom SUBU  t,ss,oo
093             SUBU  t,t,ll
094             AND   t,t,lring_mask
095             PBNZ  t,1F              Условный переход, если(rS−rO)/8 ≢ rL
                                        (по модулю ρ).
096             GET   $0,rJ             Обана, это не концевая подпрограмма.
097             PUSHJ res,StackStore    Продвинуть rS.
098             PUT   rJ,$0             Восстановить обратный адрес.
099  1H         POP   0                 Вернуться к вызывающей программе. ▮
```

Теперь перейдем к сердцевине симулятора, т.е. его основному циклу. Интерпретатор управления обычно содержит центральный раздел, который приводится в действие между интерпретируемыми инструкциями. В нашем случае, программа переходит к месту `Fetch`, когда она готова симулировать новую команду. Адрес @ новой симулируемой инструкции хранится в глобальном регистре `inst_ptr`. `Fetch` обычно задает `loc` ← `inst_ptr` и продвигает `inst_ptr` на 4. Но если нужно симулировать команду `RESUME`, которая вставляет симулированный регистр rX в поток инструкций, то `Fetch` задает `loc` ← `inst_ptr` − 4 и оставляет `inst_ptr` без изменений. Этот симулятор считает инструкцию подлежащей выполнению, только если ее адрес `loc` находится в текстовом сегменте (т.е., если `loc` < #2000000000000000).

```
100  * Основной цикл
101  loc       GREG  0   Текущее местонахождение симулятора.
102  inst_ptr  GREG  0   Следующее местонахождение симулятора.
103  inst      GREG  0   Текущая симулируемая инструкция.
104  resuming  GREG  0   Продолжить симулирование инструкции в rX?
105  Fetch  PBZ   resuming,1F      Условный переход, если нет продолжения.
106         SUBU  loc,inst_ptr,4   loc ← inst_ptr − 4.
107         LDTU  inst,g,8*rX+4    inst ← right half of rX.
```

```
108         JMP   2F
109  1H     SET   loc,inst_ptr     loc ← inst_ptr.
110         SET   arg,loc
111         PUSHJ res,MemFind
112         LDTU  inst,res,0       inst ← M4[loc].
113         ADDU  inst_ptr,loc,4   inst_ptr ← loc + 4.
114  2H     CMPU  t,loc,g
115         BNN   t,Error          Условный переход, если loc ≥ Data_Segment. ∎
```

Основной интерпретатор управления организует общий доступ к данным для всех инструкций. Он разбивает текущую инструкцию на разные части и помещает их в удобные регистры для последующего использования. Что еще более важно, он помещает 64 бита информации (“info”), соответствующей текущему опкоду, в глобальный регистр `f`. Основная таблица, которая начинается с адреса `Info`, содержит такую информацию для каждого из 256 опкодов компьютера `MMIX`. (См. табл. 1 на стр. 102.) Например, для `f` задается нечетное число, тогда и только тогда, когда поле Z текущего опкода является “немедленным” операндом или опкод является `JMP`. Аналогично, `f` ∧ #40 будет ненулевым, тогда и только тогда, когда эта инструкция имеет относительный адрес. На более поздних этапах симулятор сможет быстро решить, что нужно сделать с текущей инструкцией, поскольку большая часть необходимой информации появится в регистре `f`.

```
116  op     GREG  0   Опкод текущей инструкции.
117  xx     GREG  0   Поле X текущей инструкции.
118  yy     GREG  0   Поле Y текущей инструкции.
119  zz     GREG  0   Поле Z текущей инструкции.
120  yz     GREG  0   Поле YZ текущей инструкции.
121  f      GREG  0   Упакованная информация о текущем опкоде.
122  xxx    GREG  0   Умножение поля X на 8.
123  x      GREG  0   Операнд X и/или результат.
124  y      GREG  0   Операнд Y.
125  z      GREG  0   Операнд Z.
126  xptr   GREG  0   Адрес, по которому нужно сохранить x.
127  exc    GREG  0   Арифметические исключительные ситуации.

128  Z_is_immed_bit  IS #1    Флаговые биты, возможно в f.
129  Z_is_source_bit IS #2
130  Y_is_immed_bit  IS #4
131  Y_is_source_bit IS #8
132  X_is_source_bit IS #10
133  X_is_dest_bit   IS #20
134  Rel_addr_bit    IS #40
135  Mem_bit         IS #80

136  Info  IS    #1000
137  Done  IS    Info+8*256
138  info  GREG  Info          (Базовый адрес для основной информационной
                               таблицы.)
139  c255  GREG  8*255         (Удобная константа.)
140  c256  GREG  8*256         (Еще одна удобная константа.)

141         MOR   op,inst,#8   op ← inst ≫ 24.
142         MOR   xx,inst,#4   xx ← (inst ≫ 16) ∧ #ff.
```

```
143        MOR   yy,inst,#2  yy ← (inst ≫ 8) ∧ #ff.
144        MOR   zz,inst,#1  zz ← inst ∧ #ff.
145  0H GREG -#10000
146        ANDN  yz,inst,0B
147        SLU   xxx,xx,3
148        SLU   t,op,3
149        LDOU  f,info,t    f ← Info[op].
150        SET   x,0         x ← 0 (значение по умолчанию).
151        SET   y,0         y ← 0 (значение по умолчанию).
152        SET   z,0         z ← 0 (значение по умолчанию).
153        SET   exc,0       exc ← 0 (значение по умолчанию). ▮
```

После распаковки (т.е. разбиения) инструкции в разные поля нужно прежде всего преобразовать относительный адрес в абсолютный, если это необходимо.

```
154      AND   t,f,Rel_addr_bit
155      PBZ   t,1F              Условный переход, если это не относительный адрес.
156      PBEV  f,2F              Условный переход, если op не является JMP или JMPB.
157  9H GREG -#1000000
158      ANDN  yz,inst,9B        yz ← inst ∧ #ffffff  (а именно XYZ).
159      ADDU  t,yz,9B           t ← XYZ − 2^24.
160      JMP   3F
161  2H  ADDU  t,yz,0B           t ← YZ − 2^16.
162  3H  CSOD  yz,op,t           Установить yz ← t, если op является нечетным
                                 ("назад").
163      SL    t,yz,2
164      ADDU  yz,loc,t          yz ← loc + yz ≪ 2. ▮
```

Следующая задача очень критична для большинства инструкций: нужно установить операнды, указанные в полях Y и Z, в глобальные регистры y и z. Иногда нужно также установить третий операнд в глобальный регистр x, указанный в поле X или в специальном регистре, например в симулированном регистре rD или rM.

```
165  1H          PBNN  resuming,Install_X   Условный переход, если только
                                            не resuming < 0.
      ...                                     (См. упражнение 14.)
174  Install_X   AND   t,f,X_is_source_bit
175              PBZ   t,1F                 Условный переход, если только $X
                                            не источник.
176              SET   arg,xxx
177              PUSHJ res,GetReg
178              SET   x,res                x ← $X.
179  1H          SRU   t,f,5
180              AND   t,t,#f8              t ← номер специального регистра,
                                            умноженный на 8.
181              PBZ   t,Install_Z
182              LDOU  x,g,t                Если t ≠ 0, то x ← g[t].
183  Install_Z   AND   t,f,Z_is_source_bit
184              PBZ   t,1F                 Условный переход, если только $Z
                                            не источник.
185              SLU   arg,zz,3
186              PUSHJ res,GetReg
```

```
187             SET   z,res                 z ← $Z.
188             JMP   Install_Y
189  1H         CSOD  z,f,zz                Если Z является немедленной
                                             константой, то z ← Z.
190             AND   t,op,#f0
191             CMPU  t,t,#e0
192             PBNZ  t,Install_Y           Условный переход, только если
                                             неверно #e0 ≤ op < #f0.
193             AND   t,op,#3
194             NEG   t,3,t
195             SLU   t,t,4
196             SLU   z,yz,t                z ← yz ≪ (48, 32, 16, or 0).
197             SET   y,x                   y ← x.
198  Install_Y  AND   t,f,Y_is_immed_bit
199             PBZ   t,1F                  Условный переход, если только Y
                                             не является немедленной константой.
200             SET   y,yy                  y ← Y.
201             SLU   t,yy,40
202             ADDU  f,f,t                 Вставить Y в левую половину f.
203  1H         AND   t,f,Y_is_source_bit
204             BZ    t,1F                  Условный переход, если только $Y
                                             не источник.
205             SLU   arg,yy,3
206             PUSHJ res,GetReg
207             SET   y,res                 y ← $Y. ▮
```

Если поле X указывает выходной регистр, то для `xptr` задается адрес, по которому будет в конечном итоге хранится симулированный результат. Этот адрес будет либо в массиве `Global`, либо в кольце `Local`. Симулированный стек регистров увеличивается в этот момент, если выходной регистр меняется от маргинального до локального.

```
208  1H     AND   t,f,X_is_dest_bit
209         BZ    t,1F              Условный переход, если только $X
                                     не является выходным регистром.
210  XDest  CMPU  t,xxx,gg
211         BN    t,3F              Условный переход, если $X не является
                                     глобальным регистром.
212         LDA   xptr,g,xxx        xptr ← адрес g[X].
213         JMP   1F
214  2H     ADDU  t,oo,ll
215         AND   t,t,lring_mask
216         STCO  0,l,t             l[(α + L) mod ρ] ← 0.
217         INCL  ll,8              L ← L + 1.   ($L становится локальным.)
218         PUSHJ res,StackRoom     Убедитесь, что β ≠ γ.
219  3H     CMPU  t,xxx,ll
220         BNN   t,2B              Условный переход, если $X не является
                                     локальным регистром.
221         ADD   t,xxx,oo
222         AND   t,t,lring_mask
223         LDA   xptr,l,t          xptr ← адрес l[(α + X) mod ρ]. ▮
```

Итак, мы достигли кульминационного пункта основного цикла управления: симуляции текущей инструкции за счет 256-вариантного ветвления на основе текущего опкода. Левая половина регистра `f`, фактически, является иснтрукцией `MMIX`, которая *выполняется* в этот момент, за счет вставки ее в поток инструкций с помощью команды `RESUME`. Например, при симулировании команды `ADD` инструкция "`ADD x,y,z`" помещается в правую часть rX и очищает исключительные биты rA. Команда `RESUME` приводит к тому, что сумма регистров `y` и `z` помещается в регистр `x`, а rA зафиксирует переполнение. После выполнения команды `RESUME` управление передается к месту `Done`, если только вставленная инструкция не была условным переходом или безусловным переходом.

```
224 1H  AND   t,f,Mem_bit
225     PBZ   t,1F          Условный переход, если только inst не осуществляет
                            доступ к памяти.
226     ADDU  arg,y,z
227     CMPU  t,op,#A0      t ← [op является инструкцией загрузки].
228     BN    t,2F
229     CMPU  t,arg,g
230     BN    t,Error       Ошибка, если сохранение в текстовый сегмент.
231 2H  PUSHJ res,MemFind    res ← адрес M[y + z].
232 1H  SRU   t,f,32
233     PUT   rX,t          rX ← левая половина f.
234     PUT   rM,x          rM ← x  (подготовиться для MUX).
235     PUT   rE,x          rE ← x  (подготовиться для FCMPE, FUNE, FEQLE).
236 0H GREG #30000
237     AND   t,aa,0B       t ← текущий режим округления.
238     ORL   t,U_BIT<<8    Разрешить обход антипереполнения (см. ниже).
239     PUT   rA,t          Подготовить rA для арифметических действий.
240 0H GREG Done
241     PUT   rW,0B         rW ← Done.
242     RESUME 0            Выполнить инструкцию в rX. ∎
```

Некоторые инструкции нельзя симулировать просто "выполняя их", как команду `ADD` и переход к `Done`. Например, команда `MULU` должна вставить старшую половину вычисленного произведения в симулированный регистр rH. Команда условного перехода должна изменить `inst_ptr`, если имеет место условный переход. Команда `PUSHJ` должна протолкнуть симулированный стек регистров, а команда `POP` должна вытолкнуть его. Команды `SAVE`, `UNSAVE`, `RESUME`, `TRAP` и т.д. нужно симулировать особенно осторожно. Потому следующая часть симулятора посвящена обработке особых случаев, которые не укладываются в простую схему "`x` равно `y` операция `z`".

Начнем с достаточно простых операций умножения и деления:

```
243 MulU  MULU  x,y,z      Умножить y на z, беззнаково.
244       GET   t,rH        Установить t ← старшая половина произведения.
245       STOU  t,g,8*rH    g[rH] ← старшая половина произведения.
246       JMP   XDone       Завершить сохраняя x.
247 Div   DIV   x,y,z
    ...                     (Для деления, см. упражнение 6.) ∎
```

Если симулированная команда является условным переходом, например "`BZ $X,RA`", то основной интерпретатор управления преобразует относительный ад-

рес RA в абсолютный адрес в регистре yz (строка 164). Он также поместит содержимое симулируемого регистра $X в регистр x (строка 178). Команда RESUME затем выполнит инструкцию "BZ x,BTaken" (строка 242) и контроль будет передан BTaken вместо Done, если симулируется условный переход. BTaken прибавляет 2υ к времени симулируемого выполнения, изменяет inst_ptr и переходит к Update.

```
254  BTaken   ADDU  cc,cc,4       Увеличивает rC на 4υ.
255  PBTaken  SUBU  cc,cc,2       Увеличивает rC на 2υ.
256           SET   inst_ptr,yz   inst_ptr ← условный переход.
257           JMP   Update        Завершить выполнение команды.
258  Go       SET   x,inst_ptr    Перейти к инструкции: задать x ← loc + 4.
259           ADDU  inst_ptr,y,z  inst_ptr ← (y + z) mod 2^64.
260           JMP   XDone         Завершить сохраняя x. ▮
```

(В строке 257 мог быть переход к Done, но это было бы медленнее. Переход к Update оправдан, поскольку команда условного перехода не сохраняет x и не может привести к возникновению арифметической исключительной ситуации. См. ниже строки 500–541.)

Команда PUSHJ или PUSHGO проталкивает симулированный стек регистров за счет увеличения указателя α из (2). Это означает увеличение симулированного регистра rO, а именно регистра oo. Если команда имеет вид "PUSHJ $X,RA" и $X является локальным регистром, то X + 1 октабайтов проталкиваются вниз за счет установки $X ← X и увеличения oo на 8(X+1). (Значение в $X будет использоваться позже командой POP для восстановления oo к исходному значению. Симулированный регистр $X будет содержать результат подпрограммы, как объясняется в разделе 1.4.1′.) Если $X является глобальным регистром, rL + 1 октабайт проталкиваются вниз аналогично.

```
261  PushGo   ADDU  yz,y,z           yz ← (y + z) mod 2^64.
262  PushJ    SET   inst_ptr,yz      inst_ptr ← yz.
263           CMPU  t,xxx,gg
264           PBN   t,1F             Условный переход, если $X является
                                     локальным регистром.
265           SET   xxx,ll           Предполагается, что X = rL.
266           SRU   xx,xxx,3
267           INCL  ll,8             Увеличить rL на 1.
268           PUSHJ 0,StackRoom      Убедиться, что β ≠ γ в (2).
269  1H       ADDU  t,xxx,oo
270           AND   t,t,lring_mask
271           STOU  xx,l,t           l[(α + X) mod ρ] ← X.
272           ADDU  t,loc,4
273           STOU  t,g,8*rJ         g[rJ] ← loc + 4.
274           INCL  xxx,8
275           SUBU  ll,ll,xxx        Уменьшить rL на X + 1.
276           ADDU  oo,oo,xxx        Увеличить rO на 8(X + 1).
277           JMP   Update           Завершить выполнение команды. ▮
```

Специальные процедуры также нужны для симулирования POP, SAVE, UNSAVE, а также нескольких других опкодов, включая RESUME. Эти процедуры связаны с интересными особенностями MMIX, которые рассматриваются в упражнениях. А сейчас

мы их пропустим, поскольку они не связаны методами использования интерпретаторов, которые не рассматривались выше.

Здесь можно было бы рассмотреть код для SYNC и TRIP, но эти процедуры чересчур просты. (Действительно, для "SYNC XYZ" не нужно делать ничего, за исключением проверки XYZ ≤ 3, поскольку работа кэш-памяти не симулируется.) Вместо этого рассмотрим код для TRAP, который интересен тем, что иллюстрирует важную технологию использования таблицы переходов для многоканального (многопроходного) переключения:

```
278 Sync    BNZ   xx,Error          Условный переход, если X ≠ 0.
279         CMPU  t,yz,4
280         BNN   t,Error           Условный переход, если YZ ≥ 4.
281         JMP   Update            Завершить выполнение команды.
282 Trip    SET   xx,0              Инициировать обход к адресу 0.
283         JMP   TakeTrip          (См. упражнение 13.)
284 Trap    STOU  inst_ptr,g,8*rWW  g[rWW] ← inst_ptr.
285 0H GREG #8000000000000000
286         ADDU  t,inst,0B
287         STOU  t,g,8*rXX         g[rXX] ← inst + 2^63.
288         STOU  y,g,8*rYY         g[rYY] ← y.
289         STOU  z,g,8*rZZ         g[rZZ] ← z.
290         SRU   y,inst,6
291         CMPU  t,y,4*11
292         BNN   t,Error           Условный переход, если X ≠ 0 или
                                    Y > Ftell.
293         LDOU  t,g,c255          t ← g[255].
294 0H GREG @+4
295         GO    y,0B,y            Переход к @ + 4 + 4Y.
296         JMP   SimHalt           Y = Halt: Переход к SimHalt.
297         JMP   SimFopen          Y = Fopen: Переход к SimFopen.
298         JMP   SimFclose         Y = Fclose: Переход к SimFclose.
299         JMP   SimFread          Y = Fread: Переход к SimFread.
300         JMP   SimFgets          Y = Fgets: Переход к SimFgets.
301         JMP   SimFgetws         Y = Fgetws: Переход к SimFgetws.
302         JMP   SimFwrite         Y = Fwrite: Переход к SimFwrite.
303         JMP   SimFputs          Y = Fputs: Переход к SimFputs.
304         JMP   SimFputws         Y = Fputws: Переход к SimFputws.
305         JMP   SimFseek          Y = Fseek: Переход к SimFseek.
306         JMP   SimFtell          Y = Ftell: Переход к SimFtell.
307 TrapDone STO  t,g,8*rBB         Установить g[rBB] ← t.
308         STO   t,g,c255          Обход заканчивается с g[255] ← g[rBB].
309         JMP   Update            Завершить выполнение команды. ▌
```

(См. упражнения 15–17 для процедур SimFopen, SimFclose, SimFread, и т.д.)

Рассмотрим теперь основную таблицу Info (см. таблицу 1), которая позволяет симулятору эффективно работать с 256 разными опкодами. Каждый элемент таблицы является октабайтом, состоящим из (i) четырехбайтовой инструкции MMIX, которая вызывается инструкцией RESUME в строке 242; (ii) двух байтов, которые определяют время симулированного выполнения, один байт для μ и один байт для υ;

Таблица 1

ОСНОВНАЯ ИНФОРМАЦИОННАЯ ТАБЛИЦА ДЛЯ КОНТРОЛЯ ЗА СИМУЛЯТОРОМ

```
0 IS Done-4                                    |  LDB x,res,0; BYTE 1,1,0,#aa     (LDB)
 LOC Info                                      |  LDB x,res,0; BYTE 1,1,0,#a9     (LDBI)
 JMP Trap+@-0; BYTE 0,5,0,#0a       (TRAP)     | ...
 FCMP x,y,z; BYTE 0,1,0,#2a         (FCMP)     |  JMP Cswap+@-0; BYTE 2,2,0,#ba   (CSWAP)
 FUN x,y,z; BYTE 0,1,0,#2a          (FUN)      |  JMP Cswap+@-0; BYTE 2,2,0,#b9   (CSWAPI)
 FEQL x,y,z; BYTE 0,1,0,#2a         (FEQL)     |  LDUNC x,res,0; BYTE 1,1,0,#aa   (LDUNC)
 FADD x,y,z; BYTE 0,4,0,#2a         (FADD)     |  LDUNC x,res,0; BYTE 1,1,0,#a9   (LDUNCI)
 FIX x,0,z; BYTE 0,4,0,#26          (FIX)      |  JMP Error+@-0; BYTE 0,1,0,#2a   (LDVTS)
 FSUB x,y,z; BYTE 0,4,0,#2a         (FSUB)     |  JMP Error+@-0; BYTE 0,1,0,#29   (LDVTSI)
 FIXU x,0,z; BYTE 0,4,0,#26         (FIXU)     |  SWYM 0; BYTE 0,1,0,#0a          (PRELD)
 FLOT x,0,z; BYTE 0,4,0,#26         (FLOT)     |  SWYM 0; BYTE 0,1,0,#09          (PRELDI)
 FLOT x,0,z; BYTE 0,4,0,#25         (FLOTI)    |  SWYM 0; BYTE 0,1,0,#0a          (PREGO)
 FLOTU x,0,z; BYTE 0,4,0,#26        (FLOTU)    |  SWYM 0; BYTE 0,1,0,#09          (PREGOI)
...                                             |  JMP Go+@-0; BYTE 0,3,0,#2a      (GO)
 FMUL x,y,z; BYTE 0,4,0,#2a         (FMUL)     |  JMP Go+@-0; BYTE 0,3,0,#29      (GOI)
 FCMPE x,y,z; BYTE 0,4,rE,#2a       (FCMPE)    |  STB x,res,0; BYTE 1,1,0,#9a     (STB)
 FUNE x,y,z; BYTE 0,1,rE,#2a        (FUNE)     |  STB x,res,0; BYTE 1,1,0,#99     (STBI)
 FEQLE x,y,z; BYTE 0,4,rE,#2a       (FEQLE)    | ...
 FDIV x,y,z; BYTE 0,40,0,#2a        (FDIV)     |  STO xx,res,0; BYTE 1,1,0,#8a    (STCO)
 FSQRT x,0,z; BYTE 0,40,0,#26       (FSQRT)    |  STO xx,res,0; BYTE 1,1,0,#89    (STCOI)
 FREM x,y,z; BYTE 0,4,0,#2a         (FREM)     |  STUNC x,res,0; BYTE 1,1,0,#9a   (STUNC)
 FINT x,0,z; BYTE 0,4,0,#26         (FINT)     |  STUNC x,res,0; BYTE 1,1,0,#99   (STUNCI)
 MUL x,y,z; BYTE 0,10,0,#2a         (MUL)      |  SWYM 0; BYTE 0,1,0,#0a          (SYNCD)
 MUL x,y,z; BYTE 0,10,0,#29         (MULI)     |  SWYM 0; BYTE 0,1,0,#09          (SYNCDI)
 JMP MulU+@-0; BYTE 0,10,0,#2a      (MULU)     |  SWYM 0; BYTE 0,1,0,#0a          (PREST)
 JMP MulU+@-0; BYTE 0,10,0,#29      (MULUI)    |  SWYM 0; BYTE 0,1,0,#09          (PRESTI)
 JMP Div+@-0; BYTE 0,60,0,#2a       (DIV)      |  SWYM 0; BYTE 0,1,0,#0a          (SYNCID)
 JMP Div+@-0; BYTE 0,60,0,#29       (DIVI)     |  SWYM 0; BYTE 0,1,0,#09          (SYNCIDI)
 JMP DivU+@-0; BYTE 0,60,rD,#2a     (DIVU)     |  JMP PushGo+@-0; BYTE 0,3,0,#2a  (PUSHGO)
 JMP DivU+@-0; BYTE 0,60,rD,#29     (DIVUI)    |  JMP PushGo+@-0; BYTE 0,3,0,#29  (PUSHGOI)
 ADD x,y,z; BYTE 0,1,0,#2a          (ADD)      |  OR x,y,z; BYTE 0,1,0,#2a        (OR)
 ADD x,y,z; BYTE 0,1,0,#29          (ADDI)     |  OR x,y,z; BYTE 0,1,0,#29        (ORI)
 ADDU x,y,z; BYTE 0,1,0,#2a         (ADDU)     | ...
...                                             |  SET x,z; BYTE 0,1,0,#20         (SETH)
 CMPU x,y,z; BYTE 0,1,0,#29         (CMPUI)    |  SET x,z; BYTE 0,1,0,#20         (SETMH)
 NEG x,0,z; BYTE 0,1,0,#26          (NEG)      | ...
 NEG x,0,z; BYTE 0,1,0,#25          (NEGI)     |  ANDN x,x,z; BYTE 0,1,0,#30      (ANDNL)
 NEGU x,0,z; BYTE 0,1,0,#26         (NEGU)     |  SET inst_ptr,yz; BYTE 0,1,0,#41 (JMP)
 NEGU x,0,z; BYTE 0,1,0,#25         (NEGUI)    |  SET inst_ptr,yz; BYTE 0,1,0,#41 (JMPB)
 SL x,y,z; BYTE 0,1,0,#2a           (SL)       |  JMP PushJ+@-0; BYTE 0,1,0,#60   (PUSHJ)
...                                             |  JMP PushJ+@-0; BYTE 0,1,0,#60   (PUSHJB)
 BN x,BTaken+@-0; BYTE 0,1,0,#50    (BN)       |  SET x,yz; BYTE 0,1,0,#60        (GETA)
 BN x,BTaken+@-0; BYTE 0,1,0,#50    (BNB)      |  SET x,yz; BYTE 0,1,0,#60        (GETAB)
 BZ x,BTaken+@-0; BYTE 0,1,0,#50    (BZ)       |  JMP Put+@-0; BYTE 0,1,0,#02     (PUT)
...                                             |  JMP Put+@-0; BYTE 0,1,0,#01     (PUTI)
 PBNP x,PBTaken+@-0; BYTE 0,3,0,#50 (PBNPB)    |  JMP Pop+@-0; BYTE 0,3,rJ,#00    (POP)
 PBEV x,PBTaken+@-0; BYTE 0,3,0,#50 (PBEV)     |  JMP Resume+@-0; BYTE 0,5,0,#00  (RESUME)
 PBEV x,PBTaken+@-0; BYTE 0,3,0,#50 (PBEVB)    |  JMP Save+@-0; BYTE 20,1,0,#20   (SAVE)
 CSN x,y,z; BYTE 0,1,0,#3a          (CSN)      |  JMP Unsave+@-0; BYTE 20,1,0,#02 (UNSAVE)
 CSN x,y,z; BYTE 0,1,0,#39          (CSNI)     |  JMP Sync+@-0; BYTE 0,1,0,#01    (SYNC)
...                                             |  SWYM x,y,z; BYTE 0,1,0,#00      (SWYM)
 ZSEV x,y,z; BYTE 0,1,0,#2a         (ZSEV)     |  JMP Get+@-0; BYTE 0,1,0,#20     (GET)
 ZSEV x,y,z; BYTE 0,1,0,#29         (ZSEVI)    |  JMP Trip+@-0; BYTE 0,5,0,#0a    (TRIP)
```

Не показанные здесь элементы имеют аналогичную структуру, которую легко воспроизвести на основе представленных упражнений. (См., например, упражнение 1.)

(iii) байта-имени специального регистра, если такой регистр должен быть загружен в x в строке 182; а также (iv) байта, который является суммой восьми битовых флагов для специальных свойств опкода. Например, информация для опкода FIX имеет вид

FIX x,0,z; BYTE 0,4,0,#26;

означающая, что (i) должна быть выполнена инструкция FIX x,0,z, т.е. округление числа с плавающей точкой до числа с фиксированной точкой; (ii) время симулированного выполнения должно быть увеличено на $0\mu + 4v$; (iii) не нужен специальный регистр для входного операнда; (iv) флаговый байт имеет приведенный ниже вид, определяющий обработку регистров x, y и z:

$$\#26 = \texttt{X\_is\_dest\_bit} + \texttt{Y\_is\_immed\_bit} + \texttt{Z\_is\_source\_bit}$$

(Y_is_immed_bit означает вставку поля Y симулированной инструкции в поле Y инструкции "FIX x,0,z"; см. строку 202.)

Интересная особенность таблицы Info состоит в том, что команда RESUME в строке 242 выполняет инструкцию так, как если бы она была по адресу Done-4, поскольку rW = Done. Следовательно, если данной инструкцией является JMP, то адрес должен быть относительным к Done-4, но MMIXAL всегда ассемблирует команды JMP с адресом относительно ассемблированного положения @. Чтобы обойти ассемблер используется следующая уловка "JMP Trap+@-O", где O определяется равным Done-4. Тогда команда RESUME совершает переход к Trap в случае необходимости.

После выполнения специальной инструкции, вставленной командой RESUME, обычно происходит переход к Done. С этого места все остальное проще: инструкция успешно симулирована и текущий цикл практически завершен. Осталось оформить несколько деталей: нужно сохранить результат x в соответствующем месте, если имеется флаг X_is_dest_bit и проверить, не является арифметическая исключительная ситуация причиной прерывания обхода:

```
500 Done   AND   t,f,X_is_dest_bit
501        BZ    t,1F               Условный переход, если только $X
                                    не является местом назначения.
502 XDone  STOU  x,xptr,0           Сохранить x в симулированном регистре $X.
503 1H     GET   t,rA
504        AND   t,t,#ff            t ← новая арифметическая исключительная
                                    ситуация.
505        OR    exc,exc,t          exc ← exc ∨ t.
506        AND   t,exc,U_BIT+X_BIT
507        CMPU  t,t,U_BIT
508        PBNZ  t,1F               Условный переход, если только нет
                                    антипереполнения.
509 OH GREG U_BIT<<8
510        AND   t,aa,OB
511        BNZ   t,1F               Условный переход, если антипереполнение.
512        ANDNL exc,U_BIT          Игнорировать антипереполнение, если все
                                    точно и ситуация не активирована.
513 1H     PBZ   exc,Update
514        SRU   t,aa,8
515        AND   t,t,exc
```

```
516          PBZ   t,4F              Условный переход, если не требуется
                                     прерывание обхода.
     ...                               (См. упражнение 13.)
539  4H      OR    aa,aa,exc         Записать новые исключительные ситуации
                                     в rA. ▌
```

Строка 500 используется для удобства, хотя несколько сотен инструкций и вся таблица `Info` фактически присутствуют между строкой 309 и этой части программы. Метка `Done` в строке 500 не конфликтует с меткой `Done` в строке 137, поскольку обе они определяют эквивалентное значение для этого символа.

После строки 505 регистр `exc` содержит битовые коды для всех арифметических исключительных ситуаций, активированных только что симулированной инструкцией. В этом случае приходиться иметь дело с забавной асимметрией в правилах выполнения арифметических действий с плавающей точкой согласно стандарта IEEEю. Исключительная ситуация антипереполнения (U) подавляется, только если не активируется обход антипереполнения в rA или не возникла исключительная ситуация неточности (X). (Именно по этой причине активируется обход антипереполнения в строке 238. Симулятор завершает работу командами

```
LOC U_Handler;  ORL exc,U_BIT;  JMP Done                    (5)
```

так, что `exc` записывает исключительные ситуации антипереполнения в тех случаях, когда вычисления с плавающей точкой точны, но дают субнормальный результат.)

Наконец — Урррa! — можно завершить цикл операций, начатых с метки `Fetch`. Здесь обновляются показания часов и счетчиков, а потом происходит возврат к метке `Fetch`:

```
540  0H GREG #0000000800000004
541  Update   MOR   t,f,0B          2^32 mems + oops
542           ADDU  cc,cc,t         Увеличить показания часов, rC.
543           ADDU  uu,uu,1         Увеличить значение счетчика использования,
                                    rU.
544           SUBU  ii,ii,1         Уменьшить значение счетчика интервала, rI.
545  AllDone  PBZ   resuming,Fetch  Перейти к Fetch, если resuming = 0.
546           CMPU  t,op,#F9        В противном случае, установить
                                    t ← [op = RESUME].
547           CSNZ  resuming,t,0    Очистить resuming, если нет продолжения,
548           JMP   Fetch             и перейти к Fetch. ▌
```

Программа симулятора завершена, за исключением того, что нужно организовать правильную инициализацию. Предполагается, что симулятор запускается с помощью командной строки, в которой указан двоичный файл. В упражнении 20 описывается простой формат такого файла, в котором указывается что именно должно быть загружено в симулированную память до начала симуляции. Сразу после загрузки программы происходит следующее: в показанной ниже строке 576 регистр `loc` будет содержать адрес, в котором команда `UNSAVE` даст программе старт. (На самом деле, команда `UNSAVE` симулируется симулируемой командой `RESUME`. Для этого используется хитроумный, но работоспособный, код.)

```
549  Infile  IS    3                (Обработка двоичного входного файла.)
550  Main    LDA   Mem:head,Chunk0          Инициализировать MemFind.
551          ADDU  Mem:alloc,Mem:head,Mem:nodesize
```

```
552          GET   t,rN
553          INCL  t,1
554          STOU  t,g,8*rN          g[rN] ← (наш rN) + 1.
555          LDOU  t,$1,8            t ← имя двоичного файла (argv[1]).
556          STOU  t,IOArgs
557          LDA   t,IOArgs          (См. строку 010)
558          TRAP  0,Fopen,Infile    Открыть двоичный файл.
559          BN    t,Error
     ...                             Теперь загрузить двоичный файл
                                     (см. упражнение 20).
576          STOU  loc,g,c255        g[255] ← поместить в UNSAVE.
577          SUBU  arg,loc,8*13      arg ← поместить туда, где появляется $255.
578          PUSHJ res,MemFind
579          LDOU  inst_ptr,res,0    inst_ptr ← Main.
580          SET   arg,#90
581          PUSHJ res,MemFind
582          LDTU  x,res,0           x ← M4[#90].
583          SET   resuming,1        resuming ← 1.
584          CSNZ  inst_ptr,x,#90    Если x ≠ 0, установить inst_ptr ← #90.
585  0H      GREG  #FB<<24+255
586          STOU  0B,g,8*rX         g[rX] ← "UNSAVE $255".
587          SET   gg,c255           G ← 255.
588          JMP   Fetch             Запустить симулятор.
589  Error   NEG   t,22              t ← −22 для выхода в случае ошибки.
590  Exit    TRAP  0,Halt,0          Завершить симуляцию.
591   LOC Global+8*rK; OCTA -1
592   LOC Global+8*rT; OCTA #8000000500000000
593   LOC Global+8*rTT; OCTA #8000000600000000
594   LOC Global+8*rV; OCTA #369c200400000000
```

Стартовый адрес `Main` симулируемой программы находится в симулированном регистре $255 после выполнения симулированной команды `UNSAVE`. В строках 580–584 реализуется компонент, который не упоминался в разделе 1.3.2′: если инструкция загружается по адресу `#90`, то программа запускается с того места, а не с метки `Main`. (Это позволяет инициализировать библиотечную подпрограмму до запуска пользовательской программы с метки `Main`.)

В строках 591–594 симулированные регистры rK, rT, rTT и rV инициализируются соответствующими постоянными значениями. Затем программа завершается инструкциями обработчика обхода (5).

Уф! Программа симулятора получилась довольно большой — фактически, больше любой другой программы в этой книге. Но несмотря на большой размер, она не является полной в некоторых аспектах, поскольку автору не хотелось увеличивать ее размер.

a) Несколько частей кода вынесено в упражнения.

b) Эта программа совершает условный переход к `Error` и выходит при обнаружении любой проблемы. Качественный симулятор должен различать разные типы ошибок и находить соответствующий способ их нейтрализации и восстановления работоспособности программы.

c) Эта программа не собирает статистические данные, за исключением общего времени выполнения (cc) и общего количества симулированных инструкций (uu). Более полная программа могла бы, например, запоминать насколько часто совершались переходы по сравнению с вероятностями переходов, записывать количество обращений подпрограмм StackLoad и StackStore к симулированной памяти, анализировать собственные алгоритмы, например изучая эффективность самоорганизующейся технологии поиска на основе MemFind.

d) Эта программа не имеет никаких диагностических процедур. Например качественный симулятор, должен иметь средства интерактивной отладки и выводить избранные данные о выполнении симулированной программы. Такие средства не так уж и трудно создать, а способность отслеживать работу программы является одним из основных достоинств интерпретаторов.

УПРАЖНЕНИЯ:

1. [*20*] В табл. 1 приведены элементы только для избранных опкодов. Как будут выглядеть аналогичные элементы для опкодов (a) #3F (SRUI)? (b) #55 (PBPB)? (c) #D9 (MUXI)? (d) #E6 (INCML)?

▶ **2.** [*26*] Сколько времени потребуется симулятору для симулирования инструкций (a) ADDU $255,$Y,$Z; (b) STHT $X,$Y,0; (c) PBNZ $X,@-4?

3. [*23*] Объясните, почему $\gamma \neq \alpha$, когда StackRoom вызывает StackStore в строке 097.

▶ **4.** [*20*] Оцените критически то, что MemFind никогда не проверяет, имеет ли alloc очень большое значение. Насколько серьезно это упущение?

▶ **5.** [*20*] Если подпрограмма MemFind совершает условный переход к Error, то стек регистров не выталкивается. Сколько элементов может быть в стеке регистров в этот момент?

6. [*20*] Создайте код для симулирования инструкций DIV и DIVU, пропущенный в строках 248–253.

7. [*21*] Создайте код для симулирования инструкции CSWAP.

8. [*22*] Создайте код для симулирования инструкции GET.

9. [*23*] Создайте код для симулирования инструкции PUT.

10. [*24*] Создайте код для симулирования инструкции POP. *Замечание:* если нормальное выполнение POP, как описано в разделе 1.4.1′, приводит к rL > rG, то MMIX вытолкнет элементы из верхней части стека регситров так, что rL = rG. Например, если пользователь проталкивает 250 регистров вниз с помощью PUSHJ и выполняет “PUT rG,32; POP”, то останутся только 32 из протолкнутых вниз регистров.

11. [*25*] Создайте код для симулирования инструкции SAVE. *Замечание:* SAVE проталкивает локальные регистры вниз и сохраняет в памяти весь стек регистров, за которым следуют $\$G$, $\$(G+1)$, ..., \$255, потом rB, rD, rE, rH, rJ, rM, rR, rP, rW, rX, rY и rZ (именно в таком порядке), а потом октабайт $2^{56}\mathrm{rG} + \mathrm{rA}$.

12. [*26*] Создайте код для симулирования инструкции UNSAVE. *Замечание:* самая первая симулируемая инструкция UNSAVE является частью исходного процесса загрузки (см. строки 583–588), поэтому она не должна обновлять показания часов.

13. [*27*] Создайте код для симулирования прерывания обхода, пропущенный в строках 517–538.

14. [*28*] Создайте код для симулирования инструкции RESUME. *Замечание:* если rX неотрицательно, то наиболее значимый байт называется “ропкод” (“ropcode”). Ропкоды 0,

1, 2 доступны для пользовательских программ. В строке 242 симулятора используется ропкод 0, который просто вставляет младшую половину rX в поток инструкций. Ропкод 1 действует аналогично, но инструкция в rX выполняется с y ← rY и z ← rZ вместо обычных операндов. Этот вариант допускается только, если первой шестнадцатеричной цифрой вставленного опкода является #0, #1, #2, #3, #6, #7, #C, #D или #E. Ропкод 2 задает $X ← rZ и exc ← Q, где X является третьим байтом справа в rX и Q является третьим байтом слева. Это позволяет задать значение регистра и одновременно охватить любое подмножество арифметических исключительных ситуаций DVWIOUZX. Ропкоды 1 и 2 можно использовать только, если $X не является маргинальным. В решении упражнения следует использовать RESUME для установки resuming ← 0, если симулированный регистр rX отрицателен, либо для установки resuming ← (1, −1, −2) для ропкодов (0, 1, 2). Следует также создать отсутствующий код для строк 166–173.

▸ **15.** [*25*] Создайте код процедуры SimFputs, которая симулирует вывод строки в файл с соответствующим идентификатором.

▸ **16.** [*25*] Создайте код процедуры SimFopen, которая открывает файл с соответствующим идентификатором. (Симулятор может использовать тот же идентификатор, что и пользовательская программа.)

▸ **17.** [*25*] В продолжение предыдущих упражнений создайте код процедуры SimFread, которая считывает заданное количество байтов из файла с соответствующим идентификатором.

▸ **18.** [*21*] Будет ли полезен этот симулятор, если размер кольца локальных регистров lring_size был бы меньше 256, например lring_size = 32?

19. [*14*] Изучите все способы применения процедуры StackRoom (а именно, в строке 218, строке 268 и ответе к упражнению 11). Можете ли вы предложить более удачный способ организации кода? (См. п. 3 обсуждения в конце раздела 1.4.1′.)

20. [*20*] Входные двоичные файлы симулятора состоят из одной или более групп октабайтов, каждый из которых имеет простой вид

$$\lambda,\ x_0,\ x_1,\ \ldots,\ x_{l-1},\ 0$$

для некоторого $l \geq 0$, где x_0, x_1, …, x_{l-1} ненулевые; что означает

$$\mathrm{M}_8[\lambda + 8k] \ \leftarrow\ x_k, \qquad \text{for } 0 \leq k < l.$$

Файл заканчивается после последней группы. Завершите работу над симулятором, создав код MMIX для загрузки таких входных данных (строки 560–575 программы). Последним значением регистра loc должен быть адрес последнего загруженного октабайта, а именно $\lambda + 8(l-1)$.

▸ **21.** [*20*] Способна ли программа-симулятор из этого раздела симулировать саму себя? Если да, то способна ли она симулировать процесс собственной симуляции? Если да, то способна ли она симулировать ⋯?

▸ **22.** [*40*] Создайте эффективную процедуру *трассировочного перехода* для MMIX. Эта программа должна записывать все передачи управления в процессе выполнения другой заданной программы в виде последовательности пар (x_1, y_1), (x_2, y_2), …, которые означают переход программы из x_1 в y_1, а затем (после выполнения инструкций в местах y_1, $y_1 + 1$, …, x_2) переход из x_2 в y_2, и т.д. [На основе этой информации данный интерпретатор способен реконструировать поток выполнения заданной программы и определить частоту выполнения каждой инструкции.]

Интерпретатор трассировки отличается от симулятора тем, что позволяет трассируемой программе занимать ее обычные места в памяти. Трассировочный переход изменяет

поток инструкций в памяти, но делает это только до той степени, которая позволяет сохранять управление. В противном случае компьютер может с максимальной скоростью выполнять арифметические инструкции и операции с памятью. Однако нужно учитывать некоторые ограничения: например трассируемая программа не должна сама себя изменять. Нужно стараться сводить к минимуму такие ограничения.