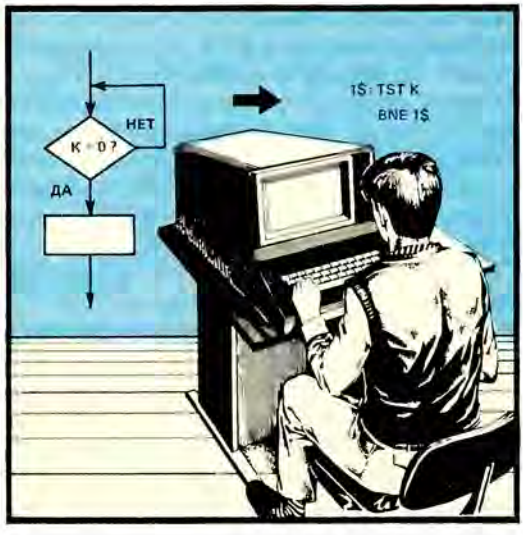


ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

Если оператор (команда) языка высокого уровня типа Бейсик или Фортран обычно транслируется в целую группу машинных команд, то сущность *Ассемблера* в символической записи отдельных команд ЭВМ. В результате этот язык полностью отражает архитектуру машины и, следовательно, позволяет создавать наиболее эффективные с точки зрения максимального использования всех возможностей ЭВМ программы.

Важная черта MACRO-11 — наличие механизма макрокоманд. Определяя *макрокоманду,* программист присваивает произвольной последовательности операторов Ассемблера некоторое имя. Появление имени в любом другом месте исходного текста приводит к автоматической подстановке данного определения на этапе трансляции программы. В отличие от подпрограммы, присутствующей в единственном числе как в исходном тексте, так и в его машинном представлении, определение макрокоманды дублируется в машинную программу столько раз, сколько оно вызывается.

Возможность использования макрокоманд, удобных в конкретной задаче или в целом классе задач, приближает MACRO-11 к проблемно-ориентированным языкам высокого уровня.

Строка программы

Исходный текст на Ассемблере записывается в виде последовательности строк, каждая из которых обычно содержит мнемоническое обозначение одной машинной команды. Для того чтобы строка исходного текста была правильно воспринята ассемблером,

необходимо соблюдать правила, совокупность которых называется *синтаксисом языка.*

Строка программы выглядит следующим образом:

МЕТКА: ОПЕРАТОР ОПЕРАНД; КОММЕНТАРИЙ, т. е. содержит 4 поля, которые могут разделяться дополнительными пробелами для того, чтобы сделать программу удобочитаемой, например:

LABEL: ADD R0, R1; ПРИБАВИТЬ R0 к R1

Поле метки, комментарий и операторная часть могут опускаться, причем в произвольных сочетаниях.

Операторы Ассемблера делятся на 3 группы:

1) *описательные операторы* — служат для резервирования внутри программы областей памяти с целью их последующего использования для хранения данных в процессе выполнения программы, присваивания значений переменным и константам и т. п.;

2) *исполняемые операторы* — это собственно машинные команды в мнемонической записи;

3) *управляющие операторы* (директивы) — с их помощью программист осуществляет управление самим ассемблером в процессе трансляции. Мнемоническое обозначение директив всегда начинается с символа точки.

Метка, или символический адрес может состоять из латинских букв, цифр и знака денежной единицы (¤). В процессе трансляции метка приобретает значение, равное адресу, по которому результат перевода обозначенной ею команды будет помещен в память. Это позволяет ссылаться в командах перехода и ветвлений как на численные адреса и смещения, так и на их символические эквиваленты, что гораздо удобнее.

Комментарии в строке программы могут начинаться с произвольного места после оператора или занимать всю строку. Комментарии записываются после символа «точка с запятой».

По умолчанию все числа, имеющиеся в тексте программы, считаются записанными в восьмеричной форме. Для иных представлений используют следующие обозначения:

↑ *Dn* или *n.*— десятичное число *n*,

↑ *Bn* — двоичное число *n*,

↑ *On* — восьмеричное число *n*,

↑ *Cn* — инверсия двоичного представления числа *n*.

Умолчание для восьмеричной формы можно отменить специальной директивой .RADIX.

Запись типа 'X представляет собой константу, равную коду ASCII символа X, а "X_1X_2 обозначает двухбайтовое число, содержащее коды ASCII символов X_1 (младший байт) и X_2(старший байт).

В процессе трансляции программы ассемблер использует специальную переменную, называемую *счетчиком адреса* и обозначаемую символом «.» (точка). Начальное значение счетчика адреса равно 0. В ходе ассемблирования он содержит адрес теку-

щей команды, полученной в результате трансляции. Точка может использоваться для записи адресов в программе. Так, команду перехода на два слова вперед можно записать в виде JMP .+4.

В тексте программы допускается использование *выражений присваивания* типа:

символ (переменная) = выражение.

Выражение может представлять собой число, другую переменную или их произвольную комбинацию, составленную с применением знаков арифметических действий: + (сложение), — (вычитание), × (умножение) и / (деление). Значение выражения подсчитывается слева направо. Важно помнить, что выражения присваивания — это лишь предписания для ассемблера, выполняемые исключительно на этапе трансляции программы.

В ходе трансляции производится синтаксический разбор программы с выдачей сообщений об обнаруженных ошибках. Может быть получен *листинг* — документ, содержащий всю информацию о трансляции, в том числе исходный текст и его машинное представление.

Приведем пример.

```
R50UNP  MACRO V05.00  Monday 01-Dec-86 11:34  Page 1
  1                              .TITLE  R50UNP
  2                     ;+
  3                     ;            ПРЕОБРАЗОВАНИЕ КОДА RADIX-50 В ASCII
  4                     ;  НА ВХОДЕ В П/П :
  5                     ;            R2 -> АДРЕС ASCII СТРОКИ
  6                     ;            SYMBOL, SYMBOL + 2 = 6 СИМВОЛОВ В КОДЕ RAD50
  7                     ;  РЕГИСТРЫ R0, R1 И R3 НЕ СОХРАНЯЮТСЯ
  8                     ;-
  9  000000 010446      R50UNP: MOV  R4, -(SP)    ; СОХРАНИТЬ R4
 10  000002 012704             MOV  #SYMBOL, R4; УКАЗАТЕЛЬ СИМВОЛОВ RAD50
             000000G
 11  000006 012401      1$:    MOV  (R4) +, R1    ;  ПОЛУЧИТЬ  ОЧЕРЕДНОЕ  СЛОВО
                                                      RAD50
 12  000010 012703             MOV  #50*50, R3    ;  ДЕЛИТЕЛЬ  ДЛЯ  СТАРШЕГО
                                                      СИМВОЛА
             003100
 13  000014 004767             JSR  PC, 10$       ; РАСПАКОВКА И ЗАПИСЬ СИМВОЛА
             000030
 14  000020 012703             MOV  #50, R3       ; ДЕЛИТЕЛЬ ДЛЯ СРЕДНЕГО СИМВОЛА
             000050
 15  000024 004767             JSR  PC, 10$       ; РАСПАКОВКА И ЗАПИСЬ
             000020
 16  000030 010100             MOV  R1, R0        ; ПЕРЕПИСАТЬ ПОСЛЕДНИЙ СИМВОЛ
 17  000032 004767             JSR  PC, 11$       ; ПРЕОБРАЗОВАТЬ И ЗАПИСАТЬ
             000016
 18  000036 020427             CMP  R4,#SYMBOL   +4 ; ПОСЛЕДНЕЕ СЛОВО?
             000004G
 19  000042 001361             BNE  1$            ; ПЕРЕХОД, ЕСЛИ ДА
 20  000044 012604             MOV  (SP) +, R4    ; ВОССТАНОВИТЬ R4
 21  000046 000207             RTS  PC            ; ВОЗВРАТ ИЗ П/П
 22                     ;+
 23                     ; ПРЕОБРАЗОВАНИЕ СИМВОЛА В ASCII:
 24                     ;       0 = ПРОБЕЛ        1...32 = A...Z
 25                     ;       33 = $            34     = .
```

```
  26                    ;       35 = НЕИСП.       36. . .47 = 0. . .9
  27                    ;-
  28  000050 005000  10$:       CLR   R0
  29  000052 071003             DIV   R3, R0
  30  000054 005700  11$:       TST   R0
  31  000056 001412             BEQ   23$
  32  000060 020027             CMP   R0#33
              000033
  33  000064 002405             BLT   22$
  34  000066 001402             BEQ   21$
  35  000070 062700             ADD   #22-11, R0
              000011
  36  000074 062700  21$:       ADD   #11-100, R0
              177711
  37  000100 062700  22$:       ADD   #100-40, R0
              000040
  38  000104 062700  23$:       ADD   #40, R0
              000040
  39  000110 110022             MOVB R0, (R2)+
  40  000112 000207             RTS   PC
  41         000001             .END
Errors detected: 0
```

Листинг программы на языке MACRO-11 состоит из следующих полей: 1 — заголовок, 2 — порядковые номера строк программы, 3 — значения счетчика адреса, 4 — коды команд, 5 — исходный текст программы.

10.2. Директивы и описательные операторы ассемблера

Описательные и управляющие операторы (директивы) ассемблера управляют процессом трансляции и не являются обозначениями машинных команд ЭВМ.

Примеры некоторых основных директив и их действия:

.TITLE PUSSYCAT — использовать слово PUSSYCAT в качестве заголовка листинга программы. Эта директива обычно предшествует всей программе;

.WORD ↑B111011, 10., 10,"10 — в данном месте программы резервируется некоторое количество (по числу аргументов) слов памяти, в которые заносятся указанные в директиве константы. В приведенном примере в последовательные ячейки будут занесены следующие восьмеричные числа:

000073
000012
000010
030061;

.BYTE 35,'A, 150, 40 — подобна директиве .WORD, но резервируются байты, а не слова:

040435
020150;

.BLKW 10., BLKB 5 — директивы для резервирования указанного числа слов или байтов, заполняемых нулями;

.ASCII /*str*/ — в последовательных байтах размещаются коды символов строки *str*. Вместо символов «/» строка может быть ограничена любыми другими одинаковыми символами (кроме «<» и «=»), которые не встречаются в строке. Непечатные спецсимволы можно записать по их кодам в виде <код>:

```
.ASCII /ДОБРОЕ УТРО/ <15> <12>
.ASCII ?ПОРА ВСТАВАТЬ!? <7> <7>
.ASCII *QUICK BROWN FOX*
```

Директива .ASCIZ действует подобно директиве .ASCII, но дополнительно вставляет нулевой байт в качестве завершающего. Обычно после директив .ASCII, .ASCIZ, .BYTE записывают директиву .EVEN, которая прибавляет единицу к счетчику адреса в случае его возможного нечетного значения.

Последним оператором программы должна быть директива

.END *sym*,

обозначающая конец программы и указывающая стартовый адрес запуска в символическом виде. Например:

.END START — выполнение программы начнется с команды под меткой START.

Для отмены восьмеричного и установки иного представления чисел, действующего по умолчанию, используется директива .RADIX *n*, где *n*=2, 10 или 8.

10.3. Как писать программы на Ассемблере?

Записи программы на Ассемблере должна предшествовать тщательная и детальная разработка алгоритма. Чтобы избежать путаницы при составлении даже простейших программ, удобно прежде всего составить блок-схему, а лучше — несколько блок-схем, различающихся степенью детализации алгоритма. При этом на самой первой схеме обычно изображаются общие этапы работы программы, а на последней — детально описывается каждый шаг, вплоть до отдельных машинных команд. Опыт говорит о том, что при наличии детальной блок-схемы процесс написания программы на Ассемблере существенно облегчается, ускоряется и требует лишь внимания и аккуратности.

Кроме того, исключительно важно знание структуры процессора (хотя бы в общих чертах) и машинного языка (архитектуры) используемой ЭВМ. Действительно, если программист не совсем точно представляет себе, каким образом ассемблер истолкует строку программы, переведет ее в машинный код, и как соответствующая ему команда выполняется, то вряд ли можно надеяться на правильную работу программы.

Вот несколько примеров, поясняющих сказанное. Допустим, желательно использовать индексный метод адресации с автоматическим увеличением содержимого регистра. Мнемоническое обо-

значение команды, использующей подобную операцию, могло бы выглядеть следующим образом:

```
CLRB 1000(R0)+
```

Вместе с тем указанного метода адресации не существует. К счастью, ассемблер тщательно анализирует допустимость применяемых методов адресации и информирует о совершенных ошибках. С другой стороны, оператор типа JMP R0 транслируется в команду с кодом 000100 без какого-либо сообщения, хотя нулевой метод адресации в сочетании с командой JMP недопустим и при ее выполнении вызывает прерывание по вектору 4. При необходимости передать управление по адресу, хранящемуся в R0, следует использовать метод 1: JMP (R0). Еще пример. Допустим, к содержимому РОН R0 нужно прибавить 4, а результат отправить в R1. Здесь легко допустить грубую ошибку, которая не будет зарегистрирована ассемблером:

```
MOV R0+4, R1
```

В результате трансляции получим инструкцию, эквивалентную

```
MOV R5, R1,
```

а требуемая операция правильно реализуется двумя командами:

```
MOV R0, R1
ADD # 4, R1
```

Следует обращать особое внимание на используемые режимы адресации для обращения к адресам и константам, поскольку, например, две следующие команды выполняют совершенно разные операции:

```
1) MOV # 1000,R0
2) MOV 1000,R0
```

В первом случае в регистр R0 будет записана константа 1000_8 (непосредственная адресация), а во-втором — содержимое ячейки с адресом 1000_8 (относительная адресация).

Следует четко разграничивать операции, выполняемые ассемблером в ходе трансляции, и выполняемые процессором при работе транслированной программы. Так, существует большая разница между командами

```
LOC:.WORD 0
```

и

```
CLR LOC
```

Первая из них предписывает ассемблеру зарезервировать ячейку памяти и заполнить ее константой 0, а вторая обозначает команду процессора, выполняемую во время работы программы.

Наконец, приведем текст программы, которую читателям

предлагается самостоятельно перевести на машинный язык и проанализировать:

```
        .TITLE   EXAMPLE
        .= 1000
START:  MOV      #A3,R0          ; УСТАНОВИТЬ АДРЕС СТРОКИ
        MOV      #6,R1           ; СЧЕТЧИК ЦИКЛОВ
LOOP:   MOVB     (R0)+,R2        ; ИЗВЛЕЧЬ ОЧЕРЕДНОЙ СИМВОЛ
        BIC      #177770,R2      ; ОЧИСТКА НЕНУЖНЫХ РАЗРЯДОВ
        ASL      R3              ; СДВИГ
        ASL      R3              ; СДВИГ
        ASL      R3              ; СДВИГ
        ADD      R2,R3           ; ДОБАВЛЕНИЕ ОЧЕРЕДНОЙ ТРИАДЫ
        SOB      R1,LOOP         ; ПОВТОР 6 РАЗ
        MOV      R3,(PC)         ; ПЕРЕСЫЛКА ДЛЯ ВЫПОЛНЕНИЯ
        .WORD    0               ; МЕСТО ДЛЯ КОМАНДЫ
        HALT
A3:     .ASCII   "010704"
        .END     START
```

Что содержится в регистрах R0...R4 по окончании работы программы?

10.4. **Подпрограммы**

Необходимая информация, касающаяся вызова и возврата из подпрограммы, уже изложена в главе 8. Напомним способы передачи аргументов в подпрограммы и способы пересылки результатов. Выбор метода передачи аргументов в подпрограммы зависит от количества аргументов и удобства применения того или иного метода в конкретном случае. При небольшом числе аргументов их можно передать в регистрах R0...R5 и через них же получить результат. При значительном количестве аргументов их размещают в последовательных ячейках ОЗУ и загружают в какой-либо из свободных РОН начальный адрес полученного массива.

Наиболее популярный метод — передача аргументов через стек. Перед входом в подпрограмму аргументы загружаются в стек, организованный на регистре R6, а по окончании ее работы стек очищается. Необходимо обратить внимание на то, что адрес возврата помещается в тот же стек самым «нижним» элементом, и следует позаботиться о его сохранности.

В качестве примера рассмотрим программу умножения двух целых двоичных 16-разрядных чисел без знака.

```
        .TITLE   MULT
START:  MOV      #58.,R0         ; УСТАНОВИТЬ СОМНОЖИТЕЛИ
        MOV      #30.,R1         ; В R0 И R1
        JSR      PC, MULT        ; ВЫЗОВ П/П УМНОЖЕНИЯ
        HALT
```

```
;+
;          ПОДПРОГРАММА УМНОЖЕНИЯ 16-РАЗРЯДНЫХ
;                       ЦЕЛЫХ ЧИСЕЛ
;-
MULT:   MOV     R0, R2      ; МНОЖИМОЕ -> R2
        CLR     R0
        MOV     #16.,R3     ; УСТАНОВИТЬ СЧЕТЧИК ЦИКЛА
        ROR     R1          ; СДВИГ СОМНОЖИТЕЛЯ (МЛ. БИТ R1 ->
                            ; С-БИТ ССП)
LOOP:   BCC     L           ; ПЕРЕХОД, ЕСЛИ C=0
        ADD     R2, R0      ; C=1 --> ДОБАВИТЬ МНОЖИМОЕ К R0
        CLC                 ; ОЧИСТИТЬ С-БИТ ССП
L:      ROR     R0          ; ЦИКЛИЧЕСКИЙ СДВИГ
        ROR     R1          ; ЧЕРЕЗ С-БИТ ССП
        SOB     R3, LOOP    ; ПОВТОР 16 РАЗ (SOB НЕ ИЗМЕНЯЕТ С-БИТА)
        RTS     PC          ; ГОТОВО -- ВОЗВРАТ ИЗ П/П
        .END    START
```

Результат программы представляет собой 32-разрядное число, т. е. двойное машинное слово. Умножение выполняется с помощью операций сдвига и сложения. Подпрограмма умножения, составляющая основу программы, возвращает результат в регистрах R1 (младшая часть) и R0 (старшая часть). В них же при входе в подпрограмму находятся два 16-разрядных сомножителя. Последовательность действий при умножении выглядит следующим образом (случай 6-разрядных чисел):

$58_{10} = 72_8 =$ 111010_2 (множимое)
$30_{10} = 36_8 =$ $\times\ 011110_2$ (множитель)

```
      000000   слагаемые 12-разрядного
+    111010    произведения формируются
+   111010     путем сдвига множимого
     111010
    111010
-------------
110110011002 = 1740(10).
```

10.5. **Программирование ввода-вывода**

В простейших программах, рассмотренных до сих пор, результаты размещались либо в ОЗУ, либо в РОН, а исходные данные записывались с использованием непосредственного метода адресации. На практике так поступают довольно редко, потому что обычно программы составляются для неоднократного использования, исходные данные могут меняться, а результаты необходимо напечатать или же записать во ВЗУ.

Остановимся на программировании операций ввода-вывода информации для двух простейших устройств — терминала и перфоленточной станции. Простоту операций с регистрами внешних устройств как с обыкновенными ячейками ОЗУ в ЭВМ типа ДВК наглядно демонстрирует программа вывода на экран символов, набираемых с клавиатуры. (Напомним, что для ЭВМ клавиатура и экран — это независимые устройства.)

```
1                                       .TITLE  ECHO
2                               ;+
3                               ;       ПРОГРАММА ПЕРЕСЫЛКИ ДАННЫХ С КЛАВИАТУРЫ
4                               ;       НА ЭКРАН ТЕРМИНАЛА (ЭХО-ПЕЧАТЬ)
5                               ;-
6                   177560              TKS = 177560    ;   РЕГИСТР СОСТОЯНИЯ КЛАВИАТУРЫ
7                   177562              TKB = 177562    ;   РЕГИСТР ДАННЫХ КЛАВИАТУРЫ
8                   177564              TPS =  177564   ;   РЕГИСТР СОСТОЯНИЯ ЭКРАНА
9                   177566              TPB =  177566   ;   РЕГИСТР ДАННЫХ ЭКРАНА
10
11  000000  105737  ECHO:   TSTB    @#TKS;  КЛАВИША НАЖАТА?
            177560
12  000004  100375          BPL     ECHO    ;   НЕТ – ПЕРЕХОД ОБРАТНО

13  000006  105737  LOOP:   TSTB    @#TPS;  ЭКРАН ГОТОВ?
            177564
14  000012  100375          BPL     LOOP    ;   НЕТ – ПЕРЕХОД ОБРАТНО
15  000014  113737          MOVB    @#TKB,@#TPB     ; ПЕРЕСЫЛКА БАЙТА
            177562
            177566
16  000022  000766          BR      ECHO    ;   ... И ВСЕ СНАЧАЛА
17          000000'         .END    ECHO
```

Обратите внимание на эффектный способ анализа состояния флага готовности в регистрах TKS и TPS. Если рассматривать младший байт этих регистров как некоторое число, то бит 07 представляет его знак ($0 = +$, $1 = -$) и записывается в бит N ССП при выполнении команды TSTB. Затем его значение анализируется командой BPL, которая осуществит ветвление, если бит N ССП равен 0.

В качестве следующего примера рассмотрим вывод строки текста на экран терминала.

```
            .TITLE   WRLINE
    ;+
    ;       ПРОГРАММА ПЕЧАТИ СТРОКИ ТЕКСТА
    ;           НА ЭКРАНЕ ТЕРМИНАЛА
    ;-
            TPS = 177564        ; ОПРЕДЕЛЕНИЕ АДРЕСОВ
            TPB = 177566        ; РЕГИСТРОВ ТЕРМИНАЛА
    ;
    START:  MOV     #BUFFER, R0         ; УСТАНОВИТЬ АДРЕС СТРОКИ
    LOOP:   TSTB    @#TPS               ; ЭКРАН ГОТОВ?
            BPL     LOOP                ; ЕСЛИ НЕТ -- ПЕРЕХОД ОБРАТНО
            MOVB    (R0)+,@#TPB         ; ВЫВОД ОЧЕРЕДНОГО СИМВОЛА
            TSTB    (R0)                ; СЛЕДУЮЩИЙ СИМВОЛ НУЛЕВОЙ?
            BNE     LOOP                ; ЕСЛИ НЕТ -- ПЕРЕХОД ОБРАТНО
            HALT                        ; ДА -- ОСТАНОВ
    BUFFER: .ASCIZ  "*****ПРИВЕТ*****"
            .END    START
```

Для вывода строки текста на экран программа циклически опрашивает готовность терминала к приему данных и при установленном флаге посылает очередной символ в регистр данных экрана. Регистр R0 используется в качестве указателя на очередной байт в выводимой строке. Читателям предлагается самостоятельно проанализировать работу этой программы.

Попытаемся разобраться с выводом цифровых данных. Составим программу для распечатки содержимого регистра R0 в восьмеричной форме. Взглянув на таблицу алфавитно-цифровых кодов, можно заметить, что алфавитно-цифровой код цифры в интервале 0...7 легко получить путем прибавления к ней числа 60_8. Например, если в R1 содержится число 7_8, то, прибавив к R1 60_8 и выдав его младший байт на экран, мы увидим цифру 7. Поэтому программа должна «разбить» содержимое R0 на тройки бит, каждая из которых соответствует одной восьмеричной цифре, добавить к каждой тройке 60_8 и распечатать их в нужном порядке. Программа выглядит так:

```
        .TITLE  PRINTR0
;+
;       ПРОГРАММА РАСПЕЧАТКИ СОДЕРЖИМОГО
;       РЕГИСТРА R0 В ВОСЬМЕРИЧНОМ ВИДЕ
;               НА ЭКРАНЕ ТЕРМИНАЛА
;-
        TPS = 177564            ; ОПРЕДЕЛЕНИЕ АДРЕСОВ
        TPB = 177566            ; РЕГИСТРОВ ТЕРМИНАЛА
;
START:  CLR     R1              ; R1 БУДЕТ СОДЕРЖАТЬ ВЫВОДИМЫЙ СИМВОЛ
        JSR     PC, EP2         ; ПЕЧАТЬ ПЕРВОЙ ЦИФРЫ (0 ИЛИ 1)
        MOV     #5, R2          ; СЧЕТЧИК ЦИКЛА
LOOP:   JSR     PC, EP1         ; ПЕЧАТЬ ОЧЕРЕДНОЙ ЦИФРЫ (0...7)
        SOB     R2, LOOP        ; ЦИКЛ ОКОНЧЕН?
        HALT                    ; ДА -- ОСТАНОВ
;+
;       ПОДПРОГРАММА ФОРМИРОВАНИЯ КОДА
;               И ПЕЧАТИ ЦИФРЫ
;-
EP1:    CLR     R1              ; ФОРМИРОВАНИЕ В R1
        ROL     R0              ; ОЧЕРЕДНОЙ ТРОЙКИ БИТ,
        ROL     R1              ; СООТВЕТСТВУЮЩЕЙ ОДНОЙ
        ROL     R0              ; ВОСЬМЕРИЧНОЙ ЦИФРЕ
        ROL     R1              ; (СДВИГ ЧЕРЕЗ С-БИТ ССП)
EP2:    ROL     R0              ;
        ROL     R1              ;
        ADD     #'0,R1          ; ДОБАВИТЬ К R1 КОД СИМВОЛА "0"
;+
;       ВЫВОД НА ЭКРАН СИМВОЛА ИЗ R1
;-
LOOP 1: TSTB    @#TPS           ; ЭКРАН ГОТОВ?
        BPL     LOOP1           ; ЕСЛИ НЕТ -- НАЗАД
        MOVB    R1@#TPB         ; ВЫВОД СИМВОЛА
        RTS     PC              ; ВОЗВРАТ ИЗ П/П
        .END    START
```

Попробуйте самостоятельно написать программы, осуществляющие функции, обратные последним двум программам, т. е. программу ввода текста в ОЗУ с клавиатуры терминала и программу ввода восьмеричного числа в R0.

10.6. Прерывания программы

Большинство относительно медленно работающих устройств — печатающих, ввода-вывода с перфоленты и т. п.— программируется способами, несущественно отличающимися от только что рассмотренных. Следует лишь помнить, что адреса регистров данных и состояния индивидуальны для каждого конкретного устройства. Высокоскоростные устройства внешней памяти на магнитных лентах и дисках программируются несколько иначе, потому что пересылка данных между ВЗУ и ОЗУ осуществляется с помощью прямого доступа в память (ПДП) блоками слов (чаще всего по 256). Процессор лишь указывает контроллеру ВЗУ номер дорожки и сектор магнитного диска, направление пересылки данных и начальный адрес отведенного для пересылаемой информации буфера в ОЗУ через специально выделенные для ВЗУ регистры команд и управления/состояния.

Основная программа обязана предварительно занести в используемые векторы прерываний нужные стартовые адреса подпрограмм обработки прерываний и значения ССП, и лишь затем позволить внешним устройствам осуществлять прерывания. Для разрешения/запрещения прерываний принято использовать разряд 06 регистров состояния внешних устройств.

Приведем пример программы дублирования перфолент, работающих с использованием прерываний.

```
        .TITLE  INTREX

;+
;       ДЕМОНСТРАЦИОННАЯ ПРОГРАММА, ПОЯСНЯЮЩАЯ
;       ПРОГРАММИРОВАНИЕ ВУ С ИСПОЛЬЗОВАНИЕМ
;       ПРЕРЫВАНИЙ ПРОГРАММЫ
;-
        PRS = 177550        ; РЕГИСТР СОСТОЯНИЯ СЧИТЫВАТЕЛЯ
        PRD = 177552        ; РЕГИСТР ДАННЫХ СЧИТЫВАТЕЛЯ
        PPS = 177554        ; РЕГИСТР СОСТОЯНИЯ ПЕРФОРАТОРА
        PPD= 177556         ; РЕГИСТР ДАННЫХ ПЕРФОРАТОРА
        PRV = 70      ; АДРЕС ВЕКТОРА ПРЕРЫВАНИЙ СЧИТЫВАТЕЛЯ
        PPV = 74      ; АДРЕС ВЕКТОРА ПРЕРЫВАНИЙ ПЕРФОРАТОРА
;
START:  MOV     #PPISR,@#PPV        ; ЗАНЕСЕНИЕ ИНФОРМАЦИИ
        MOV     #PRISR,@#PRV        ; В ВЕКТОРЫ ПРЕРЫВАНИЙ
        CLR     @#PPV+2             ;
        CLR     @#PRV+2             ;
        MOV     #101,@#PRS          ; РАЗРЕШЕНИЕ ПРЕРЫВАНИЙ
                                    ; И "СТАРТ" СЧИТЫВАТЕЛЯ
```

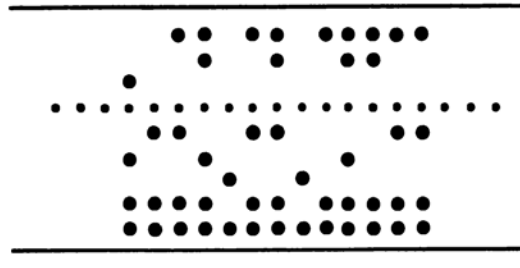
```
                    •                   ;
                    •                   ; ОСНОВНАЯ ПРОГРАММА
                    •                   ;
                                        ;
; +
;       ПРОГРАММА ОБСЛУЖИВАНИЯ ПРЕРЫВАНИЙ СЧИТЫВАТЕЛЯ
; –

PRISR:  MOVB  @#PRB,@#PPB,      ; ПЕРЕСЫЛКА БАЙТА
        MOV     #100,@#PPS      ; РАЗРЕШЕНИЕ ПРЕРЫВАНИЙ
                                ; ПЕРФОРАТОРА
        RTI                     ; ВОЗВРАТ ИЗ ПРЕРЫВАНИЯ
; +
;       ПРОГРАММА ОБСЛУЖИВАНИЯ ПРЕРЫВАНИЙ ПЕРФОРАТОРА
; –
PPISR:  CLR   @#PPS         ;ЗАПРЕТ ПОВТОРНЫХ ПРЕРЫВАНИЙ
        MOV     #101,@#PPS      ; РАЗРЕШЕНИЕ ПРЕРЫВАНИЙ
                                ; СЧИТЫВАТЕЛЯ, "СТАРТ"
        RTI                     ; ВОЗВРАТ ИЗ ПРЕРЫВАНИЯ
        .END    START
```

Перфолента представляет собой бумажную полосу, данные на которую записываются путем пробивки групп отверстий, расположенных определенным образом (рис. 10.1). Перфорирование или считывание осуществляют последовательно по 1 строке за цикл, т. е. по 1 байту данных. Порядок работы с регистрами перфоленточной станции почти такой же, как и в случае терминала, и отличается лишь тем, что для считывания очередного байта данных с перфоленты необходимо подать команду «пуск фотосчитывателя» путем установки в «1» младшего бита регистра состояния считывателя.

Использование прерывания — основной метод работы ВУ в сложных программных системах. В этом случае подпрограммы обслуживания ВУ можно оформить в виде самостоятельных модулей, выполняющихся асинхронно (т. е. независимо во времени), по отношению к основному модулю. Такие модули управления работой ВУ называются *драйверами*.

Рис. 10.1. Участок перфоленты.



10.7. **Взаимодействие программ с ОС**

Обычно программы выполняются «в среде» операционной системы (ОС). Иными словами, в момент выполнения программы помимо нее в ОЗУ ЭВМ, как правило, присутствует монитор ОС, одна из функций которого — управление вводом-выводом. Поэтому при работе с ОС в приведенных ранее примерах все манипуляции с регистрами ВУ следует заменить на обращения к ОС. Такого рода обращения осуществляются с помощью специальной *команды программного прерывания* EMT в строго определенном порядке. Вместе с тем, запоминать порядок обращения к ОС для получения того или иного обслуживания нет необходимости благодаря мощному аппарату системных макрокоманд Ассемблера. Системная макрокоманда — это совокупность машинных команд, предварительно определенная в специальной (системной) библиотеке и вызываемая автоматически по имени. Перед использованием макрокоманд в программе должна быть директива

.MCALL *имя 1, имя 2, имя 3, ...,*

которая объявляет, что указанные в ней имена являются именами макрокоманд, описанных в системной макробиблиотеке.

Для работы с терминалом в ОС ДВК существуют макрокоманды .TTYOUT и .TTYIN, предназначенные для вывода на экран и ввода с клавиатуры одного байта данных.

В качестве параметра при вызове этих макрокоманд указывают символический адрес или обозначение РОН, куда следует направить или откуда взять пересылаемый байт. Кроме того, при работе с ОС команду HALT следует заменить макрокомандой .EXIT, которая передает управление монитору ОС по окончании выполнения программы. В противном случае произойдет останов процессора и потребуется перезагрузка ОС для продолжения работы.

Пример программы для работы в среде ОС:

```
            .TITLE   OUTSTR
;+
;           ПРОГРАММА ВЫВОДА СТРОКИ ТЕКСТА НА ЭКРАН ТЕРМИНАЛА
;-
            .MCALL  .TTYOUT,.EXIT       ; ВЫЗОВ МАКРОКОМАНД
START:      MOV     #MSG,R0             ; НАЧАЛЬНЫЙ АДРЕС СООБЩЕНИЯ
            MOV     #ENDMSG-MSG,R1      ; ДЛИНА СООБЩЕНИЯ
LOOP:       .TTYOUT (R0)+               ; ВЫВОД СИМВОЛОВ
            SOB     R1,LOOP             ; В ЦИКЛЕ
            .EXIT
MSG:        .ASCII  <15><12>"ВВОД-ВЫВОД В ОС ДВК"
ENDMSG:
            .END    START
```

Конкретный набор и имена макрокоманд строго индивидуальны для каждой ОС. Так, в многопользовательской ОС RSX-11M для всех операций ввода-вывода используется универсальная макрокоманда QIOW$.

Перемещение и компоновка. Позиционно-независимое кодирование

Как уже отмечалось, этапу выполнения программы предшествует ее компоновка. *Компоновщик* осуществляет две основные функции:

1) модифицирует адреса и константы для «настройки» двоичной программы на конкретные физические адреса ОЗУ;

2) согласует взаимные обращения между различными модулями.

Необходимость в модификации адресов возникает потому, что исходная программа на Ассемблере всегда транслируется с учетом предполагаемого размещения с адреса 0. В процессе компоновки каждый объектный модуль получает некоторый участок адресного пространства, и все адреса, а также некоторые константы модифицируются с учетом базового адреса этого участка. В ряде случаев один модуль может обращаться к ячейкам, находящимся в другом. При этом метка соответствующей ячейки должна определяться в одном и другом модуле как *«глобальная»* с помощью директивы:

.GLOBL *метка 1, метка 2, метка 3, ...*

Все необходимые модификации компонуемых двоичных модулей производятся компоновщиком автоматически, без какого-либо вмешательства программиста. Используя специальные приемы программирования, можно составлять программы, не требующие модификации при загрузке в разные участки памяти. Совокупность таких приемов называется *позиционно-независимым кодированием.*

Отладка программы

Отладка — это процесс обнаружения и устранения ошибок в программе. Лишь в редких случаях вновь написанная программа сразу же начинает работать. Но это еще не означает, что она написана правильно: ошибка может проявиться позднее при каком-то определенном сочетании входных данных (скрытая ошибка). Зная это, многие опытные программисты исходят из правила: «Любая программа содержит ошибки». Можно только порадоваться вместе с читателем, которому посчастливится неоднократно опровергнуть это утверждение, но чаще всего избежать ошибок удается лишь в весьма простых программах. И дело не только в том, что программисту нужно определенное время, чтобы научиться правильно выражать свои мысли (точнее — алгоритмы)

на языке, все-таки далеком от естественного. Основная причина ошибок, допускаемых даже опытными программистами, состоит в невозможности предсказать поведение сложной разветвленной программы во всех мыслимых ситуациях и при любых наборах исходных данных. Такие ошибки — настоящий бич программирования, а цена их может быть очень велика. Известны случаи, когда в результате единственного неверного оператора срывались запуски космических кораблей, происходили серьезные аварии на производстве и транспорте. Поэтому разработка методов, облегчающих процесс обнаружения ошибок и позволяющих проверять правильность программ,— это одна из центральных проблем информатики.

Существует ряд рекомендаций, снижающих вероятность появления ошибок в программах:

— использовать по мере возможности языки высокого уровня;

— разбивать программу на отдельные законченные модули, предусмотрев возможность индивидуальной отладки каждого модуля;

— широко использовать готовые отлаженные подпрограммы из библиотек;

— избегать запутанных, излишних ветвлений, различных «трюков», призванных, например, сэкономить несколько слов в ОЗУ;

— использовать ясные и точные комментарии в программе.

Все ошибки, встречающиеся в программах, подразделяются на логические, синтаксические и опечатки, причем лишь синтаксические ошибки и в некоторых случаях опечатки могут быть обнаружены автоматически программой-транслятором. Наибольшие трудности представляет обнаружение логических ошибок, для чего существует целый ряд методов. Например, в текст программы можно временно включить специальные отладочные операторы или подпрограммы, позволяющие вывести на печать промежуточные результаты в критических точках выполнения программы, а по окончании отладки удалить их.

Как правило, каждый транслятор имеет какие-либо отладочные средства. Так, для отладки программ на языке Фортран существует специальный отладочный объектный модуль FDT (Fortran Debugging Technigue), подключаемый на этапе компоновки к отлаживаемой программе. При работе с Бейсиком для отладки используют точки останова и операторы, выполняемые в непосредственном режиме, а для отладки ассемблерных программ существует несколько модулей-отладчиков, из которых «классическим» является ODT (On-line Debugging Technique).

Вместо заключения: достижения и перспективы микропроцессорной техники

«Если бы за последние 25 лет авиационная промышленность развивалась столь же стремительно, как и вычислительная тех-

ника, то Боинг-767 можно было бы приобрести за 600 долл. и облететь на нем земной шар за 20 мин, израсходовав при этом 19 л горючего»,— эта цитата из журнала «В мире науки» как нельзя лучше отражает беспрецедентные темпы развития вычислительной техники, прежде всего — микропроцессорной. Действительно, с 1971 г. (год выпуска первого микропроцессора — Intel 4004) при снижении цен на освоенные промышленностью микропроцессоры в десятки и сотни раз количество транзисторов на кристалле возросло примерно в 200 раз, а производительность — в сотни и даже тысячи раз. Поэтому современный микропроцессорный персональный компьютер обладает вычислительной мощностью большей, чем огромный вычислительный центр 50-х годов при энергопотреблении меньшем в десятки тысяч раз.

За время создания этой книги на смену рассмотренному в ней микропроцессору К1801ВМ1 пришли его более мощные «собратья» — ВМ2 и ВМ3. О возможностях последнего можно судить хотя бы по размеру адресуемой памяти — 1920 Кслов. Эти МП — основа новых ЭВМ ДВК-3 и ДВК-4, немаловажное достоинство которых — наличие развитых средств для работы с графической информацией.

Вкратце рассмотрим другие важные достижения микропроцессорной техники и перспективы ее развития. Своеобразным «законодателем мод» остается фирма Intel (США). Восьмиразрядный МП Intel 8080, появившийся еще в 1974 г., широко применяется по сей день, а его усовершенствованные варианты (Intel 8085, Motorola 6800, Mostek 6502, Zilog 80) — основа массовых восьмиразрядных компьютеров, таких, как MSX фирмы Yamaha, ZX — Spectrum фирмы Sinclair, Apple-II, «Агат», «Ириша», «Микроша», «Корвет», Robotron-1715 и других.

Не меньший успех среди шестнадцатиразрядных МП имеют модели Intel 8086, 8088, 80186, 80286, в которых воплощено множество интересных идей. Например, они содержат небольшую встроенную память для команд, организованную по типу очереди. В то время как операционный блок выполняет текущую операцию, блок связи с магистралью автоматически извлекает из ОЗУ следующую команду и ставит ее в очередь на выполнение. Такой «конвейерный» принцип считывания и выполнения команд ощутимо повышает производительность МП. Еще одна особенность — возможность подключения отдельных БИС математических сопроцессоров, позволяющих в несколько раз повысить скорость вычислений. На МП этого семейства создаются самые популярные на сегодняшний день персональные компьютеры IBM PC и их отечественные аналоги — ЕС 1840, ЕС 1841, «Искра-1030», «Нейрон».

Одно из наиболее впечатляющих достижений — создание 32-разрядного МП Intel 80386. Уже сейчас на его основе налажен выпуск «старших» моделей нового семейства персональных компьютеров фирмы IBM — так называемых персональных систем PS/2. Разработчики утверждают, что своим вычислительным возможностям МП 80386 и первый МП 4004 соотносятся друг с другом

приблизительно так же, как космический корабль «Шаттл» и аэроплан братьев Райт.

А что же последует за 32-разрядными микропроцессорами? Специалисты склонны считать, что напрашивающегося перехода к 64-разрядным МП не произойдет. Действительно, «большие» ЭВМ уже более 40 лет остановились на 32 разрядах и похоже, что такой длины слова вполне хватает для большинства задач. Можно ожидать появления 64-разрядных шин между процессором и памятью, но по существующим оценкам развитие МП скорее всего пойдет по путям совершенствования их архитектуры. Одно из направлений — создание МП с сокращенным набором команд, или процессоров типа RISC (Reduced Instruction Set Computer). Сторонники этого направления расценивают усложнение обычных МП как чрезмерное, поскольку из сотен команд, которые «умеет» выполнять типичный современный МП, в среднем сколь-либо интенсивно используется лишь небольшая их часть. Таким образом, 80—90 % аппаратуры МП в среднем бездействует. Кстати, самая первая массовая мини-ЭВМ PDP-8, созданная еще в 60-х годах, имела лишь 8 основных команд. Типичный RISC-процессор рассчитан лишь на десятки команд, каждая из которых выполняется за один машинный цикл, а не за несколько, как у обычных МП. Выпускаемые в настоящее время десятки различных моделей 16- и 32-разрядных RISC-МП отличаются повышенным быстродействием (до 100 млн операций в с), высокой технологичностью изготовления, а следовательно, высокой надежностью и меньшей стоимостью.

Другое направление привело к созданию уникальных приборов, получивших название «транспьютер». Транспьютер — это однокристальная ЭВМ, содержащая процессор (обычно RISC), память, интерфейсы внешней памяти и периферии, а также средства, обеспечивающие многозадачный режим работы и взаимодействие нескольких транспьютеров. В числе этих средств — скоростные последовательные линии связи (обычно — 4), которые позволяют объединять множество транспьютеров в ячеистые структуры с параллельной обработкой данных. Потенциальные возможности структур из множества транспьютеров трудно переоценить, причем препятствием на пути их широкого распространения является не техническая реализация, а разработка математического обеспечения параллельной обработки данных.

Транспьютерный суперкомпьютер — уже факт. На очереди — реализация новых архитектур из транспьютеров. Может быть, за ними — будущее вычислительной техники. Однако не будем торопиться с прогнозами, поскольку практика нередко перечеркивает даже самые правдоподобные из них.