МАЛЫЕ ЭВМ ВЫСОКОЙ ПРОИЗВОДИТЕЛЬНОСТИ

Архитектура и программирование

Москва
«Радио и связь»
1990

ББК 24.4.4
М 20
УДК 681.322—181.4.004.15

Авторы: Г. П. ВАСИЛЬЕВ, Г. А. ЕГОРОВ, В. С. ЗОНИС, М. А. ОСТРОВСКИЙ, В. В. РОДИОНОВ, К. В. ПЕСЕЛЕВ

Рецензенты: кандидаты техн. наук А. В. Никаноров, Я. П. Локшин

Редакция литературы по информатике и вычислительной технике

М20 **Малые** ЭВМ высокой производительности. Архитектура и программирование / Г. П. Васильев, Г. А. Егоров, В. С. Зонис и др.; Под ред. Н. Л. Прохорова.— М.: Радио и связь, 1990.— 256 с.: ил.

ISBN 5-256-00316-X.

Рассматриваются архитектура и характеристики 32-разрядных мини-ЭВМ СМ1700. Приводятся структура этой ЭВМ, методы представления информации, режимы адресации и система команд, принципы организации ввода-вывода и виртуальной памяти, язык ассемблера и основные функции операционной системы. Оцениваются показатели производительности с учетом архитектурных особенностей ЭВМ, уточняются области ее применения в комплексных интегрированных АСУ.

Для пользователей СМ ЭВМ, системных программистов и инженеров-разработчиков технических и программных средств СМ ЭВМ.

М $\frac{2404040000\text{-}056}{046\,(01)\text{-}90}$ 141-90 **ББК 24.4.4**

Производственное издание

ВАСИЛЬЕВ ГЕННАДИЙ ПЕТРОВИЧ, ЕГОРОВ ГЕННАДИЙ АЛЕКСЕЕВИЧ, ЗОНИС ВЛАДИМИР СЕМЕНОВИЧ, ОСТРОВСКИЙ МИХАИЛ АЛЕКСАНДРОВИЧ, РОДИОНОВ ВЛАДИМИР ВЛАДИМИРОВИЧ, ПЕСЕЛЕВ КОНСТАНТИН ВИКТОРОВИЧ

МАЛЫЕ ЭВМ ВЫСОКОЙ ПРОИЗВОДИТЕЛЬНОСТИ. АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ

Заведующая редакцией *Г. И. Козырева* Редакторы *С. Н. Удалова, В. И. Ченцова*
Переплет художника *В. Е. Самохина* Художественный редактор *А. В. Проценко*
Технический редактор *З. Н. Ратникова* Корректор *Т. Л. Кускова*

ИБ 1721

Сдано в набор 9.08.89 Подписано в печать 31.01.90 Т- 06033 Формат $60\times88^{1}/_{16}$ Бумага офсетная № 2 Гарнитура «Таймс» Печать офсетная Усл. печ. л. 15,68 Усл. кр.-отт. 15,68 Уч.-изд. л. 18,17 Тираж 25 000 экз. Изд. № 22832 Зак. № 2840 Цена 1 р. 20 к

Издательство «Радио и связь», 101000 Москва, Почтамт, а/я 693

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО «Первая Образцовая типография» Государственного комитета СССР по печати. 113054, Москва, Валовая, 28.

ISBN 5-256-00316-X

ПРЕДИСЛОВИЕ

В книге рассматриваются вопросы архитектуры, программирования и применения 32-разрядных мини-ЭВМ семейства СМ с системным интерфейсом «Общая шина». В народном хозяйстве широко применяются ЭВМ этого семейства (16-разрядные СМ4, СМ1420, СМ1600 и др.). Многие пользователи смогли оценить как положительные свойства архитектуры этих ЭВМ, так и недостатки. Дальнейшее развитие семейства СМ привело к созданию 32-разрядных ЭВМ, первым представителем которых является СМ1700.

Основной целью создания этой ЭВМ явилось увеличение размера виртуального адресного пространства, поскольку емкость оперативной памяти, один из главных показателей, в значительной степени определяет все остальные характеристики. Семейство 16-разрядных ЭВМ позволяет непосредственно из программы адресовать 64 Кбайт, а семейство 32-разрядных — до 4 Гбайт.

Принципиальным вопросом является преемственность создаваемых 32-разрядных ЭВМ и действующих 16-разрядных. Она заключается в том, чтобы сохранить для пользователя и производителя возможность максимального использования в новом семействе основных аппаратных средств (в первую очередь системного интерфейса «Общая шина» и периферийных устройств) и программных средств (в первую очередь прикладного программного обеспечения, файлов и баз данных).

Преемственность СМ1700 и 16-разрядных ЭВМ обеспечивается:

системным интерфейсом «Общая шина» (следовательно, теми же периферийными устройствами);

форматами представления данных (все файлы могут быть прочитаны на любой ЭВМ);

сходством синтаксиса и мнемоник языка ассемблера, а также полным совпадением версий языков высокого уровня (выполнение программ, написанных на них, дает одинаковые результаты);

операционными системами.

Кроме того, центральный процессор СМ1700 имеет специальный аппаратный режим совместимости, который обеспечивает выполнение базового набора команд 16-разрядных ЭВМ, за исключением некоторых привилегированных команд. Поэтому пользовательские программы ЭВМ семейства СМ, в которых не используют привилегированные команды, могут выполняться без переработок.

Режимы адресации СМ1700 являются развитием режимов адресации СМ4, СМ1420 и др. Система команд СМ1700 призвана обеспечить эффективный машинный код, генерируемый компиляторами с различных языков программирования. Некоторые операторы языков высокого уровня и примитивы операционных систем реализованы аппаратно. Это относится, например, к способу вызова подпрограмм, реализации сложных переходов на многие направления, управлению циклами, вычислению индексов массива, обработке очередей и списков, обработке контекста процесса, преобразованию виртуального адреса и т. д.

Процессор СМ1700 широко использует принцип пользовательского динамического микропрограммирования, позволяющий ориентировать архитектуру как на универсальные, так и на конкретные задачи пользователя, что увеличивает производительность в несколько раз.

Главы 1, 7, § 8.1, 8.2 написаны Зонисом В. С., гл. 2 — Островским М. А., гл. 3, 4, приложения — Родионовым В. В., гл. 5, § 8.7 — Егоровым Г. А., гл. 6 — Васильевым Г. П., § 8.4—8.6 — Песелевым К. В., § 8.3, 8.8 — Егоровым Г. А. и Песелевым К. В.

Авторы выражают надежду, что эта книга будет с интересом встречена специалистами по вычислительной технике.

Генеральный конструктор СМ ЭВМ *Н. Л. Прохоров*

Глава 1. ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ В ЭВМ

1.1. ВВЕДЕНИЕ В АРХИТЕКТУРУ ВЫЧИСЛИТЕЛЬНОГО КОМПЛЕКСА СМ1700

Вычислительный комплекс СМ1700 представляет собой универсальную ЭВМ. Одна из основных целей его создания — это расширение виртуального адресного пространства по сравнению с ЭВМ типа СМ4. Хотя некоторые инструкции вычислительного комплекса СМ1700 имеют сходство с инструкциями ЭВМ семейства СМ4, СМ1700 представляет собой полностью новую архитектуру.

Аппаратные средства СМ1700 ориентированы на реализацию языков высокого уровня и системных программ, которые используются операционной системой и компилятором. Система команд (инструкций) СМ1700 в настоящее время включает 304 инструкции (приложения 1 и 2) и более 20 режимов адресации операндов. Все это дает возможность программисту составлять эффективные по объему и времени выполнения программы.

На рис. 1.1 показана структура вычислительного комплекса СМ1700 с дополнительными компонентами, расширяющими его возможности. Некоторые из этих компонентов, например процессор арифметики с плавающей точкой, который ускоряет выполнение соответствующих инструкций арифметики с плавающей точкой и некоторых инструкций фиксированной арифметики, не оказывают влияния на методику составления программ. Такие компоненты, как принято говорить, «прозрачны» для программ. Они представляют собой часть технических средств и будут освещены в другой главе. Здесь кратко описываются те компоненты, которые касаются программиста (подробнее см. гл. 7).

Процессор вычислительного комплекса СМ1700 является главным управляющим устройством всей системы. Он работает под микропрограммным управлением и реализует всю систему команд. Кроме того, в режиме совместимости с ЭВМ типа СМ4 он может выполнять их непривилегированные инструкции, тем самым обеспечивая исполнение пользовательских программ, написанных для семейства СМ4.

Центральный процессор содержит шестнадцать 32-разрядных быстродействующих регистров общего назначения. Некоторые из них имеют специальное назначение, а один используется как

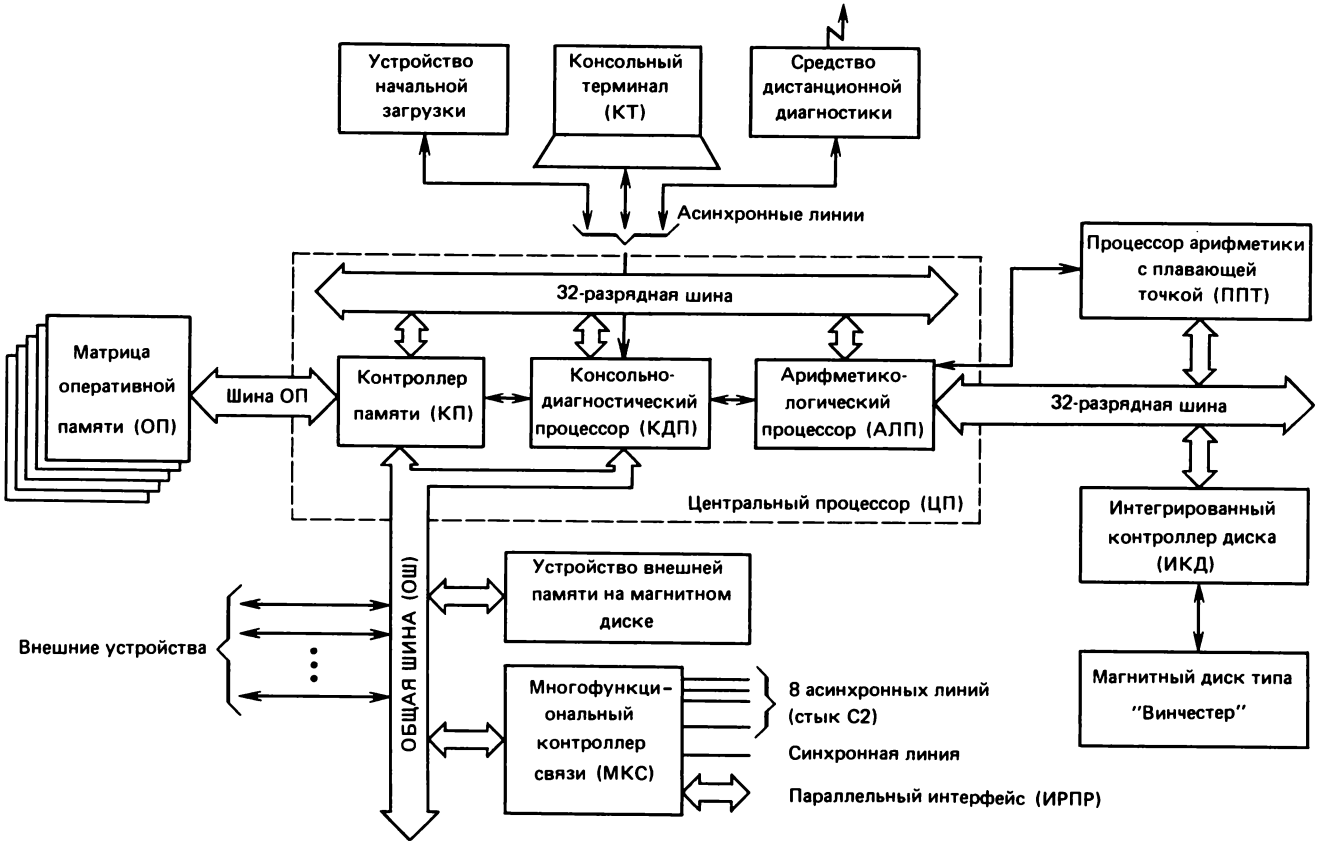


Рис. 1.1. Структура вычислительного комплекса СМ1700

счетчик инструкции. В процессоре имеется также 32-разрядное двойное слово состояния процессора, которое содержит информацию о процессоре и текущем состоянии выполняемой программы. Оно состоит из двух 16-разрядных слов. Младшее, называемое словом состояния программы, содержит информацию о программе пользователя и доступно пользователю. Старшее — привилегированную информацию о состоянии процессора и может быть модифицировано только операционной системой. Таким образом с помощью пользовательской программы можно обращаться к слову состояния программы и изменять его, но состояние процессора защищено от модификации пользователем.

Физическая оперативная память меняется в диапазоне 1...5 Мбайт. Виртуальная память преобразуется в физическую с помощью диспетчера памяти, содержащего для этих целей так называемый буфер трансляции. Связь процессора с оперативной памятью осуществляется по специальному внутреннему интерфейсу.

Процессор содержит также буфер инструкций, который введен для повышения его производительности. В этот буфер производится чтение из оперативной памяти очередной инструкции на фоне исполнения в процессоре текущей.

Вычислительный комплекс СМ1700 имеет консольно-диагностический процессор, построенный на базе специального микропроцессора, в функции которого входит первоначальная загрузка микропрограммного и микродиагностического обеспечения, управление микро- и дистанционной диагностиками, а также управление консольным терминалом и устройством первоначальной загрузки.

В качестве системного интерфейса для подключения процессора к периферийным устройствам используется «Общая шина», т. е. тот же интерфейс, что и ЭВМ семейства СМ4. Это дает возможность подключать имеющиеся у этого семейства периферийные устройства к вычислительному комплексу СМ1700.

Системный магнитный диск подключается к центральному процессору через специальный контроллер, аппаратура которого интегрирована с аппаратурой процессора. Этим достигается повышение быстродействия дисковых накопителей в режиме передачи данных и снижение загрузки системного интерфейса. Все это приводит к повышению производительности всего комплекса.

1.2. ФОРМАТЫ И ТИПЫ ДАННЫХ

В вычислительном комплексе СМ1700 адресация данных производится с точностью до байта. Основная единица информации в СМ1700 — 8-разрядный байт. Однако система инструкций обеспечивает работу с числами другого размера, сформирован-

ными из групп байтов и бит. Вычислительный комплекс СМ1700 может работать с 16-разрядными словами (два байта), 32-разрядными двойными словами (четыре байта), 64-разрядными учетверенными словами (восемь байтов). Каждая из указанных информационных единиц (рис. 1.2) сформирована из групп последовательно расположенных байтов и всегда адресуется так же, как самый младший байт в данной группе. Заметим, что разряды в любой информационной единице нумеруются с самого младшего (разряд 0), расположенного справа, до самого старшего, расположенного слева, т. е. до 7-го, 15-го, 31-го или 63-го разрядов соответственно для байта, слова, двойного слова или учетверенного слова.

Другая единица информации, используемая в СМ1700,— это битовое поле переменной длины. Оно характеризуется тем, что его длина измеряется не в байтах, а в битах. Длина поля может меняться от 0 до 32 последовательно расположенных бит и может быть расположена произвольно по отношению к начальной границе байта. Чтобы описать битовое поле, необходимо наличие трех атрибутов:

1) базового адреса A, указывающего на байт в памяти, в котором искомое поле начинается;

2) позиции первого бита P внутри выбранного байта по отношению к разряду 0;

3) размера S поля в битах.

На рис. 1.3 показано битовое поле с базовым адресом A, позицией первого бита P и размером S.

Битовые поля, вообще говоря, используются для совместной упаковки многочисленных информационных полей. Инструкции,

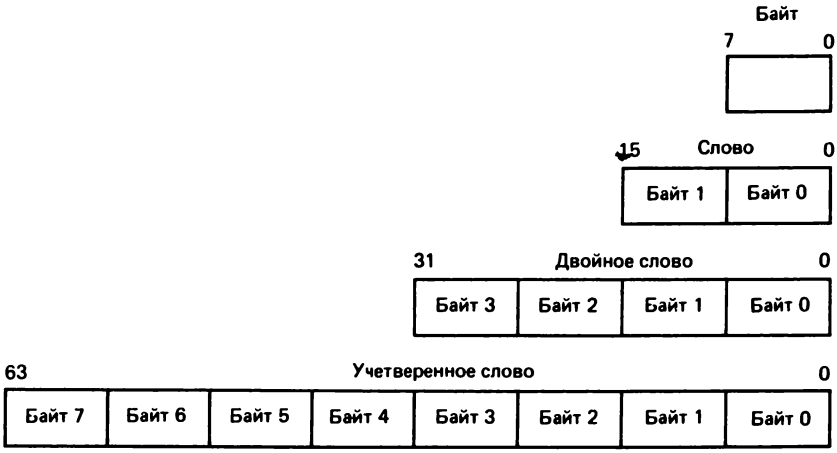


Рис. 1.2. Единицы информации вычислительного комплекса СМ1700

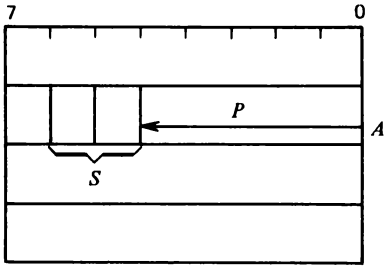


Рис. 1.3. Битовое поле переменной длины

которые работают с битовыми полями, позволяют обрабатывать поля с размерами меньше одного байта.

Байт, слово, двойное и учетверенное слова являются основными единицами информации в СМ1700. Они образуют типы данных. Все информационные единицы состоят из последовательности двоичных цифр. В свою очередь, единица информации может представлять букву алфавита, целое или действительное число и т. д. Процессор СМ1700 способен обрабатывать различные типы данных. Это позволяет программисту (или компилятору) создавать очень компактные программы, используя необходимые типы данных. Имеется также полный набор инструкций для преобразования одних типов данных в другие, что упрощает многие программы.

Целые числа. Числа, которые обрабатываются процессором вычислительного комплекса СМ1700, могут иметь формат с фиксированной и плавающей точками. Первый формат отображает целые числа. Они могут быть представлены как двоичными числами без знака, так и числами в дополнительном коде со знаком, и могут занимать размер байта, слова (два байта), двойного слова (четыре байта) или учетверенного слова (восемь байтов). Длина целого числа зависит от максимального значения числа и длины отображающей его единицы информации. В табл. 1.1 показан диапазон представления для различных целых. Вычислительный комплекс СМ1700 имеет полный набор инструкций, оперирующих со всеми типами данных для операций сложения, вычитания, умножения, деления, взятия дополнительного кода, сдвига целых чисел, хотя арифметика с учетверенными числами поддерживается не полностью.

Полный набор инструкций условного перехода позволяет программисту изменять последовательность выполнения программ, базируясь на результате выполнения предыдущей арифметической операции.

В вычислительном комплексе СМ1700 имеются инструкции для выполнения арифметических действий над целыми числами

Таблица 1.1. **Типы данных для целых чисел**

Тип целого	Размер, бит	Диапазон (в десятичной системе)	
		Числа со знаком	Числа без знака
Байт	8	От -128 до $+127$	От 0 до 255
Слово	16	От $-32\,768$ до $+32\,767$	От 0 до 65 535
Двойное слово	32	От -2^{31} до $+2^{31}-1$	От 0 до $2^{32}-1$
Учетверенное слово	64	От -2^{63} до $+2^{63}-1$	От 0 до $2^{64}-1$

без знака размером больше 64 разрядов (расширенная точность).

Числа с плавающей точкой. Целые числа удобны для представления данных и решения задач во многих областях, однако их диапазона представления бывает недостаточно. Хотя целые представляют данные с достаточной точностью, значения данных или промежуточных результатов могут превышать разрядную сетку. Числа с плавающей точкой, как и целые, представляют собой совокупность последовательно расположенных разрядов в памяти. Эта совокупность имеет две части: мантиссу (дробную часть) и порядок (экспоненту). Таким образом, число с плавающей точкой можно представлять следующим образом (рис. 1.4): $\pm 0, n \times 2^{\pm m}$, где n — мантисса, а m — порядок. Заметим, что здесь указаны знаки (S) как для мантиссы, так и для порядка. Однако более общей формой является представление с указанием только одного знакового разряда. В этом представлении самый старший значащий разряд слова есть знак мантиссы, а порядок — положительное число со смещением. Хотя порядок есть положительное число в диапазоне от 0 до $m-1$, половина этого диапазона (от 0 до $m/2$) используется для представления отрицательных чисел, а другая половина (от $m/2$ до $m-1$) — для представления положительных чисел.

Формат числа с плавающей точкой в вычислительном комплексе СМ1700 такой же, как и в СМ1420 и СМ1600. В СМ4 обрабатываются числа только одинарной точности, формат которых также совпадает с соответствующими числами в СМ1700.

Числа с плавающей точкой одинарной точности состоят из четырех последовательно расположенных байтов и имеют адрес, совпадающий с адресом байта, содержащего бит 0. Бит 15 двойного слова определяет знак числа, а 8-разрядный порядок отделяет старшую значащую часть мантиссы от знака. Младшая

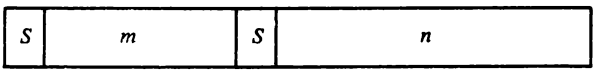


Рис. 1.4. Формат числа с плавающей точкой

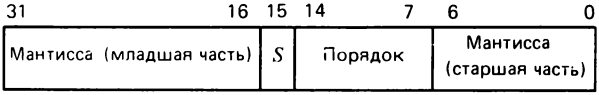


Рис. 1.5. Формат числа с плавающей точкой одинарной точности

часть мантиссы размещается в оставшихся 16 разрядах. Итак, число с плавающей точкой одинарной точности в СМ1700 имеет следующий формат (рис. 1.5).

Дробная часть представляется 24-разрядной положительной мантиссой, большей или равной 0,5 и меньшей или равной 1. Двоичная запятая расположена левее самого старшего значащего разряда. Так как этот разряд при нормализованной мантиссе всегда должен быть 1 (если число не равно нулю), то в памяти запоминается только 23-разрядная мантисса.

Порядок формируется как смещенное на 128_{10} 8-разрядное положительное целое число. Следовательно, если из порядка вычесть 128_{10}, то в результате получится действительное значение порядка. Истинное значение числа с плавающей точкой получается умножением мантиссы на степень числа, т. е. на 2^m, где m — порядок.

Знак числа будет положительным, когда знаковый разряд равен 0, и отрицательным, когда 1.

Число с плавающей точкой в 32-разрядном формате описывается следующим уравнением:

$$X=(1-2\times S)\times \text{мантисса}\times 2^{(\text{порядок}\ -128)},$$

где

$$2^{-128}=2{,}939\times 10^{-39}\leqslant X\leqslant 2^{127}=1{,}701\times 10^{38}.$$

Поле мантиссы в 32-разрядном формате представляет 7 десятичных цифр, а в 64-разрядном формате (числа с плавающей точкой двойной точности) 16 десятичных цифр. Число с двойной точностью требует для своего представления восемь последовательно расположенных байтов.

Кроме чисел одинарной и двойной точности в СМ1700 могут быть представлены числа двойной точности с расширенным порядком (64 разряда или 8 байт) и учетверенной точности с расширенным порядком (128 разрядов или 16 байт).

Заметим, что если порядок и знак числа равны нулю, то число X считается также равным нулю независимо от значения мантиссы.

Алфавитно-цифровые символы. Помимо целых чисел и чисел с плавающей точкой вычислительному комплексу СМ1700 приходится оперировать с алфавитно-цифровыми символами. Ал-

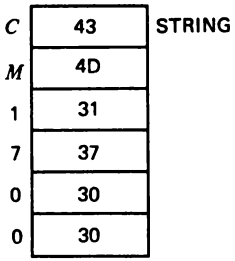


Рис. 1.6. Представление в 8-разрядном коде СМ1700 строки «СМ1700»

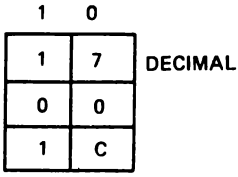


Рис. 1.7. Представление десятичной строки для числа +17001

фавитно-цифровые символы представляются числовым кодом для каждого символа. При этом используется 8-разрядный код СМ1700 (приложение 3) для обмена информацией. Набор символов включает в себя буквы алфавита (латинские и русские), цифры (0...9), знаки препинания и специальные управляющие знаки. Этот код предназначен для обеспечения связи между ЭВМ и периферийным оборудованием различных производителей.

На рис. 1.6 показано представление в коде СМ1700 алфавитно-цифровой строки «СМ1700». Укажем, что каждый символ размещается в одном байте, который представляется двумя шестнадцатеричными цифрами. Обращение к этой строке указывается адресом первого байта, байта, содержащего символ «С» и имеющего символический адрес «STRING».

В СМ1700 имеются инструкции для работы со строками из последовательно расположенных байтов. Эти инструкции называются инструкциями обработки символьных строк. Символьная строка имеет два атрибута: адрес и длину в байтах (или символах). Она может быть длиной от 0 до 65 635 байт.

Десятичные строки. Числа могут представляться в ЭВМ как строки десятичных цифр. Обычно числа вводятся в ЭВМ в коде СМ1700, а затем для выполнения арифметических операций преобразуются в двоичную форму. Однако в ряде случаев удобнее работать не с двоичной формой, а с десятичной. В частности, это имеет большое значение для коммерческих задач (при работе с языком Кобол), где требуется выполнять только простые вычисления.

Вычислительный комплекс СМ1700 работает с двумя формами десятичных строк. В первой форме, которая называется числовой строкой, каждая десятичная цифра занимает один байт. Числовые строки могут иметь разный формат в зависимости от того, где расположен знаковый разряд: перед первой цифрой или после последней. Вторая форма, называемая упакованной десятичной, компонует две десятичные цифры в один байт.

Таблица 1.2. **Представление упакованного десятичного символа**

Цифра или знак	Десятичное значение	Шестнадцатеричное значение	Цифра или знак	Десятичное значение	Шестнадцатеричное значение
0	0	0	6	6	6
1	1	1	7	7	7
2	2	2	8	8	8
3	3	3	9	9	9
4	4	4	+	12	C
5	5	5	−	13	D

В табл. 1.2 дано представление упакованного десятичного символа. Упакованная десятичная строка составляется из последовательности байтов в памяти, причем каждый байт разделен на две тетрады (тетрада — это 4-разрядное поле). Каждая тетрада представляет одну десятичную цифру. Последняя тетрада в строке десятичного числа предназначена для указания знака. Например, упакованная десятичная строка для числа +17001 показана на рис. 1.7.

Десятичная строка определяется двумя атрибутами: адресом первого байта в строке (обозначаемом как DECIMAL на рис. 1.7) и длиной строки, измеряемой числом тетрад (в данном случае шесть), т. е. десятичных цифр, включая знак (C). В СМ1700 имеются инструкции для выполнения арифметических операций над десятичными строками и для преобразования десятичных

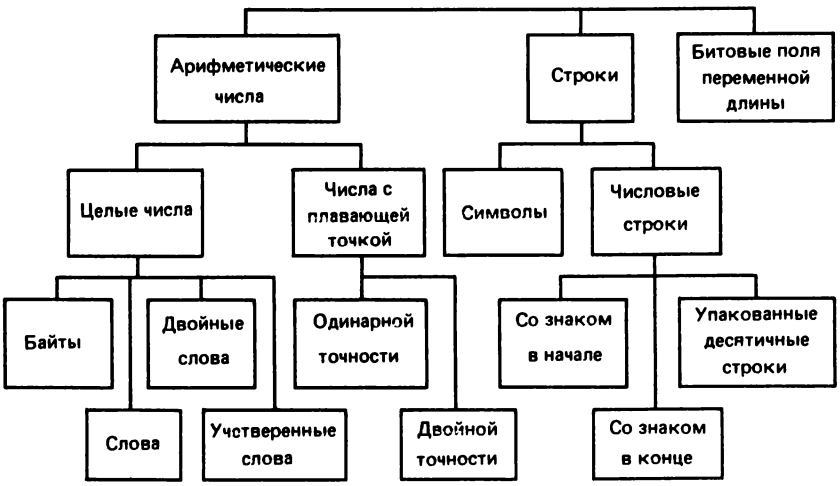


Рис. 1.8. Типы данных, используемые в вычислительном комплексе СМ1700

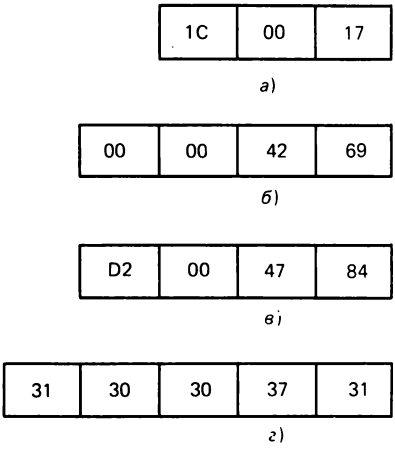


Рис. 1.9. Представления числа 17001

строк в другие форматы. Чтобы произвести операции над десятичными числами, нужно ввести их в коде СМ1700, затем преобразовать в упакованную десятичную форму, выполнить необходимые операции и сделать обратное преобразование в код СМ1700 для вывода информации.

Обзор используемых типов данных. Выше были рассмотрены способы представления информации в ЭВМ. Хотя в оперативной памяти вся информация хранится в двоичном виде, интерпретировать ее можно по-разному, и в зависимости от этого используются соответствующие машинные инструкции. Выбор нужной интерпретации зависит от применения ЭВМ. На рис. 1.8 показаны типы данных, используемые в СМ1700.

Мы можем взять одну и ту же информацию и представить ее различным образом. Рисунок 1.9 иллюстрирует, как число 17001 может быть представлено четырьмя различными способами: в виде упакованной десятичной строки (*а*), целым числом в формате двойного слова (*б*), числом с плавающей точкой одинарной точности (*в*) и в виде строки в коде ЭВМ СМ1700 (*г*). Каждое представление дано в виде строки байтов (в шестнадцатеричной форме).

Так же, как и буквы алфавита, можно кодировать машинные инструкции. К этим инструкциям пришлось бы обращаться с помощью их двоичного кода, если бы не было символьного ассемблера, который позволяет использовать имена, или мнемоники, вместо кодов. Кодирование инструкций в СМ1700 рассмотрено в последующих главах, а сейчас необходимо понять основные инструкции в их символьной форме.

1.3. ФОРМАТЫ ИНСТРУКЦИЙ И ФУНКЦИИ АССЕМБЛЕРА

В языках программирования высокого уровня, таких, например, как Паскаль, Фортран и другие, программы представляются в виде алгебраических выражений с использованием служебных слов. Во всех ЭВМ, работающих с этими языками, имеется программа трансляции языка, называемая компилятором, которая считывает программу на языке высокого уровня (называемую исходной) и транслирует ее в программу на машинном языке

(называемую объектной). Так как понятия и операторы исходного языка не могут выполняться ЭВМ напрямую, компилятор для каждого исходного оператора языка генерирует несколько машинных инструкций.

Одно из главных преимуществ языков высокого уровня (помимо простоты описания алгоритмов) — их машинная независимость, т. е. независимость от технических средств ЭВМ и их систем команд. Тем не менее имеется ряд причин, обусловливающих необходимость программирования на машинном языке. Во-первых, это дает возможность лучше учитывать архитектуру и особенности конкретной ЭВМ. Понимание того, как работает аппаратура, позволяет писать более качественные программы. Во-вторых, имеется ряд задач, связанных с управлением специфическим оборудованием данной ЭВМ, для которых требуется машинный уровень программирования. Например, та часть операционной системы, которая управляет памятью и устройствами ввода-вывода, обычно пишется на языке машинного уровня. Если поставлена цель понять, как работает ЭВМ, то необходимо знание машинного языка.

Для облегчения программирования на машинном языке используется язык ассемблера, в котором для обозначения машинных операций используются символьные обозначения инструкций и ячеек памяти.

Формат инструкций вычислительного комплекса СМ1700. Инструкция в СМ1700 содержит код операции и от нуля до шести спецификаторов операндов. Формат инструкции на языке ассемблера имеет вид: МЕТКА: КОД ОПЕРАЦИИ,СПЕЦИФИКАТОР1,СПЕЦИФИКАТОР2,...;КОММЕНТАРИЙ. Здесь МЕТКА — набор алфавитно-цифровых символов, который служит для обозначения адреса инструкции; КОД ОПЕРАЦИИ — мнемоника или символьное имя инструкции в вычислительном комплексе СМ1700; СПЕЦИФИКАТОР — символьное имя или указатель для адреса операнда; КОММЕНТАРИЙ — словесное описание функций, выполняемых инструкцией.

Метку и комментарий указывать не обязательно, а число требуемых спецификаторов определяется кодом операции.

Подробное описание инструкций в вычислительном комплексе СМ1700 и режимов адресации дается в гл. 2, 3. В табл. 2.1—2.4 приводится простое подмножество инструкций вычислительного комплекса СМ1700. Эти инструкции, наиболее часто используемые начинающими программистами, включают одно-, двух- и трехадресные инструкции.

Чтобы понять, как инструкции работают, будет полезно рассмотреть инструкцию вычитания двойных слов. В инструкции ячейки операндов, над которыми проводится операция вычитания, будут указываться их адресами. Например, инструкция SUBL 1000,1004 вычитает двойное слово, являющееся содержимым

ячейки 1000 из содержимого ячейки 1004. Результат двухадресных арифметических инструкций всегда запоминается по адресу второго операнда. Таким образом, содержимое ячейки 1004 модифицируется инструкцией.

Предположим, что ячейки 1000 и 1004 содержат следующие значения перед выполнением инструкции SUBL:

Адрес	Содержимое
1000	12
1004	2578
1008	324

После выполнения инструкции SUBL 1000,1004 содержимое ячеек памяти будет следующим:

Адрес	Содержимое
1000	12
1004	2566
1008	324

Некоторые арифметические инструкции вычислительного комплекса СМ1700 могут иметь три операнда. Если нужно сохранить содержимое ячеек 1000 и 1004 и запомнить результат в ячейке 1008, то необходимо записать SUBL3 1000,1004,1008.

После выполнения этой инструкции содержимое ячеек будет таким:

Адрес	Содержимое
1000	12
1004	2578
1008	2566

Обычно адреса указываются с помощью символьных имен ячеек. Числовые значения адресов определяются ассемблером в процессе трансляции программы. Символьные имена (метки) указывают на определенные строки в теле программы, и это может быть указание либо на строку инструкций, либо на ячейку данных.

В макроязыке вычислительного комплекса СМ1700 (т. е. языке ассемблера) обозначение LABEL: в начале оператора является его именем и отражает его адрес. Макроязык имеет также операторы-директивы, так называемые псевдооператоры, которые не генерируют инструкции в программе, а размещают ячейки памяти и определяют их содержимое. Например, оператор E: .WORD 11 заставляет ассемблер выделить 16-разрядную ячейку памяти и записать туда число 11, а оператор E: .LONG 11 — 32-разрядную ячейку с тем же содержимым. Символ **E** является адресом этой ячейки. С помощью символьных меток предыдущий пример с инструкцией вычитания может быть записан в следующем виде:

```
Адрес          Оператор ассемблера

1000           A: LONG 12
1004           B: LONG 2578
1008           C: LONG 324
  .              .
  .              .
  .              .
2000             SUBL3 A,B,C
```

Адреса, указанные слева, приведены только для иллюстрации и не определяются программистом. Ассемблер берет на себя всю работу по присвоению и запоминанию адресов, предоставляя возможность программисту работать только с их символьными именами.

Важно помнить, что символы A, B и C представляют адреса трех двойных слов, а не содержимое этих ячеек. Значение символа A в приведенном примере есть 1000, в то время как значение двойного слова, запомненного в ячейке с адресом 1000, есть 12. Это отличается от концепции символа A в большинстве языков высокого уровня, где символы и его значение — синонимы. При программировании на ассемблере всегда нужно помнить о различии между адресом и данными, записанными по этому адресу.

Функции символьного ассемблера. Как уже говорилось, ассемблер вычислительного комплекса СМ1700 выполняет задачу по переводу программы с символьного языка ассемблера в программу на двоичном машинном языке. При переводе программы ассемблер выполняет следующие функции:

1) генерирует символьную таблицу значений всех определенных пользователем идентификаторов (символьных имен и меток);

2) следит за счетчиком ячеек, который указывает, где разместится следующая инструкция или данные в оперативной памяти;

3) переводит символьные имена инструкций, спецификаторы операндов, идентификаторы и выражения в двоичный машинный код (называемый объектной программой);

4) выдает распечатку программисту, в которой указываются все инструкции и данные программы, а также адреса, по которым они разместились в ячейках памяти.

Чтобы понять, как работает ассемблер, полезно изучить механизм трансляции, который включает формирование счетчика ячеек, определение и использование идентификаторов, распределение памяти, вычисление выражений и использование управляющих операторов.

Идентификаторы. Идентификаторы представляют собой последовательность алфавитно-цифровых символов. Эта последовательность может состоять из букв алфавита, цифр от 0 до 9 и специальных символов : точки «.», знака подчеркивания «_» и знака $. Максимальная длина последовательности символов составляет 15 знаков и не должна начинаться с цифры.

Идентификатор может быть определен двумя способами. Первый способ определения — метка. Например, исходная строка DOCTOR: ADDL MIND,WILL вводит метку DOCTOR. Ассемблер вычислит адрес DOCTOR при трансляции этой инструкции.

Второй способ определения — прямое присвоение с использованием знака равенства «=». Например, оператором

```
EIGHT=8
LETTERCOUNT=40
MATRIXSIZE=23
```

будут присвоены значения, указанные справа. Такие присвоения удобны при программировании. Никакого машинного кода они не генерируют.

Как метки с символом «:» на конце, так и прямые присвоения с символом «=» имеют локальный характер, т. е. действительны только для данной программы. Чтобы придать им глобальный характер за пределами данной программы, используются соответственно знаки :: и ==.

Константы. Ассемблер интерпретирует все константы как десятичные целые числа. Однако программист может указать константу в другой системе счисления так, как это показано в табл. 1.3. Например, оператор COUNT_OF_NUMBS =^X5A присвоит левой части значение 5A в шестнадцатеричной форме.

Чтобы определить константу в кодах СМ1700, нужно перед константой и после указать ограничительные символы. Например, во всех трех случаях

```
ZON=^A/ZON/
ZON=^A.ZON.
ZON=^AXZONX
```

будет определена одна и та же константа ZON.

Счетчик ячеек. При записи программ в оперативной памяти обычно исходят из предположения, что последовательно расположенные инструкции записываются в последовательно расположенных ячейках памяти. Хотя имеются операторы и инструкции, которые изменяют последовательность исполнения программы, программисты при составлении программы предполага-

Таблица 1.3. **Унарные операторы ассемблера**

Оператор	Представление	Оператор	Представление
^B	Двоичное	^X	Шестнадцатеричное
^D	Десятичное	^A	Символьное
^O	Восьмеричное	^F	Формат с плавающей запятой

ют, что инструкции выполняются последовательно, т. е. большинство инструкций не имеют поля адреса, который бы указывал адрес следующей инструкции, и, следовательно, они должны записываться в ячейках памяти по порядку. Позднее, когда программа исполняется, счетчик инструкций должен автоматически увеличиваться после чтения текущей инструкции с тем, чтобы указывать на следующую, которая будет выполняться после завершения текущей. Аналогично счетчику инструкций ассемблер имеет счетчик ячеек для отслеживания места размещения следующего байта инструкции или данных.

В ассемблере обычно принято, что первый байт программы будет размещаться в ячейке с адресом «0». По мере того, как ассемблер читает инструкции и данные из исходной программы, он транслирует их и выдает на выходе байты объектной программы. Счетчик ячеек инкрементируется после выдачи каждого байта. Таким образом, он всегда содержит адрес, по которому будет записан следующий байт.

В ассемблере вычислительного комплекса СМ1700 «.» является символом счетчика ячеек. При использовании ассемблерной директивы (из перечня операторов-директив) он представляет адрес текущего байта. Например,

```
VLADIM: .LONG .;выделение ячейки VLADIM
```

дает указание ассемблеру разместить адрес ячейки VLADIM в самой ячейке VLADIM (длина которой должна быть равна двойному слову). Вообще говоря, предпочтительнее пользоваться метками, а не указанием адресов через счетчик ячеек, так как окончательную программу легче читать. При использовании точки необходимо проявлять внимательность, поскольку небрежность ее написания может привести к ошибкам. Так, операторы

```
CHIP: .LONG  CHIP
      .LONG  CHIP

CHIP: .LONG  .
      .LONG  .
```

не эквивалентны. В первом случае оба двойных слова содержат адрес CHIP. Во втором случае второе удвоенное слово содержит свой собственный адрес CHIP+4.

Распределение памяти. Распределение памяти для данных производится операторами-директивами, или ассемблерными директивами, некоторые из которых перечислены в табл. 1.4. С помощью этих директив можно выделять ячейки памяти размером в байт, слово, двойное слово и т. д., а также распределять большие блоки памяти. Например, оператор DIGIT: .BYTE 5,6,7 запоминает целые числа 5, 6, 7 в трех последовательно расположенных ячейках памяти длиной в байт каждая, адресом первой ячейки является метка DIGIT.

Таблица 1.4. **Операторы-директивы распределения памяти**

Директива	Назначение
Метка: .BYTE список значений	Запоминает указанные в директиве значения в последовательно расположенных ячейках памяти длиной в байт каждая. Символьным адресом первой ячейки является «метка», как и в последующих операторах-директивах
Метка: .WORD список значений	Запоминает указанные значения в последовательно расположенных ячейках памяти длиной в слово каждая
Метка: .LONG список значений	Запоминает указанные значения в последовательно расположенных ячейках памяти длиной в двойное слово каждая
Метка: .QUAD список значений	Запоминает указанные значения в последовательно расположенных ячейках памяти длиной в учетверенное слово каждая
Метка: .ASCII /строка/	Запоминает заключенную в ограничителях строку символов в кодах СМ1700 в последовательно расположенных ячейках памяти длиной в байт каждая
Метка: .ASCIC /строка/	То же, что и предыдущий оператор, только с добавлением в конце строки байта, содержащего число символов в строке
Метка: .ADDRESS список	Запоминает указанные 32-разрядные адреса ячеек памяти
Метка: .BLKB размер	Резервирует область памяти указанного размера из ячеек длиной в байт. Символьным адресом первой ячейки является «метка». Все резервируемые ячейки обнуляются
Метка: .BLKW размер	Резервирует область памяти указанного размера из ячеек длиной в слово
Метка: .BLKL размер	Резервирует область памяти указанного размера из ячеек длиной в двойное слово
Метка: .BLKQ размер	Резервирует область памяти указанного размера из ячеек длиной в учетверенное слово

Оператор INTEGER: .LONG 1,765,318,10 запоминает целые числа 1, 765, 318, 10 в четырех последовательно расположенных ячейках памяти длиной в двойное слово каждая.

Для выделения больших блоков памяти без указания содержимого ячеек в блоке используются директивы .BLKX, где X обозначает длину ячейки в блоке.

Операторы

```
MATRIXSIZE=23
MATRIX: .BLKL MATRIXSIZE
```

выделяют 23 последовательно расположенные ячейки размером в двойное слово. Поскольку это удобно для ассемблера, все ячейки в блоке обнуляются.

Выражения. Это комбинации термов, соединенных бинарными операторами (табл. 1.5). Терм может быть или числом, или символом.

Таблица 1.5. **Бинарные операторы ассемблера**

Оператор	Пример использования	Функция
+	$A+B$	Сумма A и B
−	$A-B$	Вычитание из терма A терма B
×	$A\times B$	Произведение A и B
/	A/B	Целое частное от деления A на B
@	$A@B$	Арифметический сдвиг A на B разрядов. Если B положительное, то A сдвигается влево, а в младший разряд вдвигается 0. Если B отрицательное, то A сдвигается вправо, при этом старший разряд повторяет свое значение, которое последовательно сдвигается вправо
&	$A\&B$	Логическое И над A и B
!	$A!B$	Логическое ИЛИ над A и B
\\	$A\backslash\backslash B$	Исключающее ИЛИ над A и B

Выражения и все термы вычисляются как 32-разрядные числа. Выражения обрабатываются слева направо без учета приоритета операторов, однако угловые скобки $\langle\ \rangle$ могут изменять порядок вычисления. Например, $A+B\times C$ и $A+\langle B\times C\rangle$ имеют разные значения.

Угловые скобки могут вкладываться внутрь других скобок, как в выражении $\langle\langle A\&B\rangle+C\rangle$.

Выражения могут использоваться в разных местах программы. Ниже приводятся примеры использования выражений:

```
. WORD  17*<15-10>
. BLKL  MATRIXSIZE*5
HEIGHT=5*6*MATRIXSIZE
```

Знак * используется в качестве знака умножения.

Управляющие операторы. Рассмотрим четыре наиболее часто используемых в этой книге ассемблерных управляющих оператора.

Оператор заголовка

```
.TITLE имя модуля ;комментарий
```

определяет имя программного модуля и содержит строку комментариев. Заголовок печатается на верхней строчке каждой страницы распечатки.

Оператор подзаголовка

```
.SBTTL ;комментарий
```

содержит строку комментариев, которая должна быть напечатана на второй строчке каждой страницы распечатки. Оператор .TITLE является всегда первой строкой модуля, в то время как оператор .SBTTL используется перед каждой подпрограммой внутри модуля. Ассемблер также использует оператор подзаголовка, чтобы создать таблицу содержимого на первой странице распечатки.

Оператор программной секции

.PSECT имя секции

может использоваться (помимо других целей) и для разделения инструкций и данных. Так же, как операторы заголовка и подзаголовка служат для разделения модулей программ и подпрограмм, оператор программной секции служит для разделения программ на сегменты. Каждый сегмент, таким образом, является частью программы с присущим ему набором меток и символом. Во время фазы компоновки программные сегменты объединяются и между ними разрешаются взаимные ссылки.

И, наконец, оператор конца

.END метка

информирует ассемблер, что является последней строкой, которая должна быть обработана. Метка оператора .END указывает ассемблеру начальный адрес оттранслированной программы. Когда программа должна выполняться, она будет начинаться в указанной ячейке. Метка этого оператора является меткой внутри программы (главной). Если модуль содержит только подпрограммы, то эта метка опускается.

Распечатка. На рис. 1.10 показана типичная распечатка, полученная с помощью макроассемблера. Каждая строка в ней содержит следующую информацию:

1. Десятичный номер строки для ссылок.

```
                                        0000  100
                                   000C 0000  200
                              0000000A  0002  250
                        50   0000'8F    3C  0002  300
                    54   00000000'EF    DE  0007  500
                              56   01   D0  000E  600
                         57   01   0A   C3  0011  700
                              58   0A   D0  0015  800 START:
                         59   56   01   C1  0018  900
                       FE A448   6448   D1  001C 1000 PROCE:
                                   17   18  0022 1100
                  00000014'EF    6448   B0  0024 1200
                         6448    FE A448  B0  002C 1300
               FE A448    00000014'EF    B0  0032 1400
                                         01  003B 1500 CONT1:
                                   58    D7  003C 1600
                              59   58    D1  003E 1650
                                   D9    18  0041 1675
                             CE 36   S7    F2  0043 1700 CONT2:
                                         01  0047 1800
                                         01  0048 1900 RESULT:
                                         00  0000 2000
0020 0043 0001 0007 0005 002B 000A       0000 2100 AREA:
                         0003 0008 0004  000E
                                   0000  0014 2300 RAB:
                                         0016 2400
```

Рис. 1.10. Распечатки программы, полу

2. Символьную строку из исходной символьной программы.

3. Шестнадцатеричный адрес, по которому размещается первый байт сгенерированной в машинном коде информации соответствующей символьной строке по п. 2 (представляет собой смещение от начала программной секции).

4. Шестнадцатеричное представление данных, размещенных в оперативной памяти, а также сгенерированный код операции для инструкции (причем адреса байтов размещаются справа налево).

Ассемблер печатает также символьную таблицу в конце распечатки, указывая в нем значение (или адрес) для каждого идентификатора.

Ассемблирование (трансляция). Теперь, когда имеется первое представление о синтаксисе и семантике языка ассемблера, кратко опишем процесс ассемблирования (трансляцию программы). Он становится понятен при изучении простого двухпроходного ассемблера, который читает исходную программу дважды. Два прохода обусловлены тем, что перед тем, как процесс трансляции может фактически начаться, должны быть известны значения всех идентификаторов.

Цель ассемблера — прочитать исходную программу на символьном языке пользователя и создать двоичную объектную программу и распечатку.

Цель первого прохода заключается в построении символьной таблицы всех идентификаторов, встречающихся в программе. Например, символьная таблица для программы, приведенной на рис. 1.10, будет содержать информацию, приведенную в табл. 1.6.

```
.TITLE  SORT                ;Урок 1
.ENTRY  FIRST,^M<R2,R3>
N=10                        ;Счетчик
MOVZWL  *SS$NORMAL,R0
MOVAL   AREA,R4             ;В R4 область адресов
MOVL    #1,R6               ;Начало первого индекса
SUBL3   #N,*1,R7            ;Конец первого индекса
MOVL    #N,R8               ;Начало второго индекса
ADDL3   #1,R6,R9            ;Конец второго индекса
CMPL    (R4)[R8],-2(R4)[R8] ;Нужно изменить?
BGEQ    CONT1               ;Не нужно
MOVW    (R4)9[R8],RAB
MOVW    -2(R4)[R8],(R4)[R8]
MOVW    RAB,-2(R4)[R8]
NOP
DECL    R8
CMPL    R8,R9
BGEQ    PROCE
AOBLSS  R7,R6,START     ;Продолжить внешний цикл
NOP
NOP
.PSECT  DATA
.WORD   10,43,5,7,1,67,32,4,8,3

.WORD
.END    FIRST
```

ченной с помощью макроассемблера

Таблица 1.6. **Символьная таблица идентификаторов**

Наименование	Значение (шестнадцатеричное)	Тип	Наименование	Значение (шестнадцатеричное)	Тип
AREA	0	Метка	PROCE	1C	Метка
CONT1	3B	»	RAB	14	»
CONT2	43	»	RESULT	48	»
FIRST	0	Идентифи-	START	15	»
N	A	катор			

Первый проход ассемблера начинается с установки счетчика ячеек в ноль. После чтения каждой строки проводится ее синтаксический анализ и определение типа оператора. Если строка содержит метку, то ей присваивается текущее значение счетчика ячеек и она запоминается в символьной таблице. Значения вычисленных идентификаторов размещаются в символьной таблице. Если в очередной строке встречается инструкция или оператор, выделяющий какую-то область памяти, то ассемблер увеличивает счетчик ячеек на размер инструкции или размеры выделяемой памяти для данных.

Во время второго прохода ассемблер использует символьную таблицу для трансляции программы в двоичный код. Он начинается также с обнуления счетчика ячеек и чтения исходной программы. Ассемблер компонует для каждой строки исходной программы строку для распечатки, которая состоит из исходного текста, номера строки, счетчика ячеек (адреса), шестнадцатеричного представления для инструкций и данных. При чтении каждая исходная строка снова проверяется на синтаксис, и замеченные ошибки распечатываются. Метки и символьные определения обычно проверяются в соответствии с символьной таблицей. При чтении команды ассемблер использует символьную таблицу для нахождения значений спецификаторов символьных операндов (идентификаторов). Инструкция, содержащая код операции и спецификаторы, затем транслируется в двоичную форму и прикомпоновывается к объектному файлу. Если встретился оператор распределения памяти, то ассемблер выделяет в объектной программе указанный размер памяти с указанным в исходной программе содержимым.

Глава 2. ФОРМАТ ИНСТРУКЦИЙ И РЕЖИМЫ АДРЕСАЦИИ

2.1. ОБЩИЕ СВЕДЕНИЯ

В этой главе рассматриваются способы адресации данных в СМ1700. Кроме того, описываются некоторые другие архитектурные особенности, знание которых необходимо для более полного понимания структуры адресации. В отличие от комп-

лексов типа СМ4, где адресация большого объема памяти затруднена ввиду ограниченного размера разрядной сетки (16 разрядов), в вычислительном комплексе СМ1700 используется 32-разрядный адрес, который позволяет адресовать практически неограниченный объем памяти (4 Гбайт). Однако принятые в ЭВМ типа СМ4 общие принципы адресации сохранены и получили дальнейшее развитие. Это должно позволить программистам сравнительно легко освоить новую систему адресации.

Для иллюстрации режимов адресации в главе будут частично рассмотрены основные инструкции, некоторые особенности, а также представление данных. Одной из важнейших особенностей архитектуры СМ1700 является соответствие между основными типами данных и операциями, которые могут быть произведены над ними. Целые числа могут быть представлены в виде байтов (8 разрядов), слов (16 разрядов) и длинных слов (32 разряда) и имеется широкий набор инструкций, которые оперируют с каждым из этих типов данных одинаковым образом и в одной и той же последовательности. Длинное слово может представлять собой не только целое число, но и адрес. Поэтому в СМ1700 имеется ряд инструкций, которые предназначены для обработки как числовой, так и адресной информации.

2.2. ОСНОВНЫЕ ОПЕРАЦИИ

Система инструкций СМ1700 будет подробно рассмотрена в следующей главе. Однако для изучения режимов адресации потребуется знание некоторого набора инструкций, которые производят наиболее часто используемые операции.

Вычислительный комплекс СМ1700 имеет относительно симметричную систему инструкций. Это означает, что все основные инструкции могут оперировать с любыми типами данных. Поэтому мнемоника инструкций формируется из названия операций, таких как MOV (пересылка) или ADD (сложение), с добавлением суффикса, указывающего на тип операнда; B, W или L для обозначения целых чисел, представленных в виде байта, слова или двойного (длинного) слова. В некоторых случаях операнд может иметь длину 64 разряда (учетверенное слово). Тогда к мнемонике инструкции добавляется суффикс Q. Инструкции, реализующие одноадресные операции (такие как логическая инверсия, отрицание, инкрементирование и т. п.), могут иметь один или два операнда. В первом случае результат запоминается по тому же адресу, где находился и исходный операнд, во втором — по адресу второго операнда, который будем называть «приемником». Содержимое первого операнда, который будем называть «источником», остается без изменения. Инструкции, реализующие двухместные операции (такие как сложение, вычитание и т. п.), могут иметь два или три операнда.

Таблица 2.1. **Основной набор инструкций СМ1700 для работы с целыми числами**

Название	Операция	Мнемоника	Тип данных	Число операндов
Пересылка	Переслать содержимое источника по адресу приемника	MOV	B, W, L, Q	2
Очистка	Очистить содержимое приемника	CLR	B, W, L, Q	1
Пересылка с отрицанием	Переслать отрицательное значение источника по адресу приемника	MNEG	B, W, L	2
Пересылка с инверсией	Переслать поразрядную логическую инверсию содержимого источника по адресу приемника	MCOM	B, W, L	2
Инкремент	Увеличить содержимое приемника на единицу	INC	B, W, L	1
Декремент	Вычесть единицу из содержимого приемника	DEC	B, W, L	1
Сложение	В случае двухоперандной формы результат запомнить по адресу второго операнда. В случае трехоперандной формы содержимое первых двух операндов сложить, а результат запомнить по адресу третьего операнда	ADD	B, W, L	2, 3
Вычитание	В случае трехоперандной формы содержимое первого операнда вычесть из содержимого второго, а результат запомнить по адресу третьего операнда	SUB	B, W, L	2, 3

В первом случае результат запоминается по адресу второго операнда (приемника), исходное содержимое которого теряется. Для первого операнда (источника) оно остается без изменения. Во втором случае результат запоминается по адресу третьего операнда (приемника), а содержимое двух первых операндов (источников) остается без изменения. В этих инструкциях число операндов определяется добавлением цифр 2 или 3 к мнемонике инструкции после указателя типа данных. Например, инструкция сложения может иметь следующие варианты: ADDB2, ADDB3, ADDW2, ADDW3, ADDL2, ADDL3. Так как двухоперандная форма инструкций используется наиболее часто, при написании программ на ассемблере допускается пропуск суффикса 2.

В табл. 2.1 показаны основные арифметические и логические инструкции. Для каждой инструкции приводятся допустимые типы операндов и их возможное число. Инструкции пересылки и очистки могут оперировать как байтами, словами, длинными словами, так и учетверенными словами.

2.3. УПРАВЛЕНИЕ ПОРЯДКОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Одной из важнейших особенностей архитектуры любой ЭВМ является способ организации «ветвления» внутри программ, т. е.

возможности выполнять различные функции в зависимости от условий, возникающих в процессе выполнения программы. Для этой цели служат инструкции условного перехода, которые должны содержать информацию об условии перехода и адресе передачи управления. Для правильного функционирования инструкций условного перехода необходимо запоминать состояние процессора после выполнения каждой инструкции для того, чтобы следующая инструкция могла использовать информацию об этом состоянии. В ЭВМ СМ1700 имеются четыре разряда состояния процессора, которые называются кодами условий и отражают результат последней выполняемой инструкции. Эти разряды являются младшей частью регистра расширенного слова состояния процессора, содержимое которого называется «словом состояния процессора». Для определения порядка своих действий инструкции переходов используют следующие коды условий:

N — *разряд отрицательного результата.* Устанавливается в единицу, если результат выполнения инструкции был отрицательным, и сбрасывается в нуль, если положительным или нулевым.

Z — *разряд нулевого результата.* Устанавливается в единицу, если результат операции точно равен нулю, в противном случае сбрасывается в нуль.

V — *разряд переполнения.* Устанавливается в единицу, если в результате выполнения арифметической команды произошло переполнение, т. е. результат вышел за диапазон представления чисел, допустимый для используемого в инструкции типа данных.

C — *разряд переноса.* Устанавливается в единицу, если в результате выполнения арифметической инструкции происходит перенос из старшего значащего разряда или заимствование.

В общем случае каждая инструкция имеет свой собственный способ установки кодов условий. Некоторые инструкции устанавливают или сбрасывают коды условий в соответствии с результатом выполненной операции, другие — безусловным образом, остальные не оказывают на эти коды никакого влияния. Приведенные выше определения относятся к наиболее часто используемым способам установки кодов условий. Разница между переполнением и переносом не всегда очевидна. Поэтому рассмотрим этот вопрос более подробно. Каждый байт, слово или длинное слово данных могут представлять собой целое со знаком соответственно в диапазонах $-2^7 \ldots 2^7-1$, $-2^{15} \ldots 2^{15}-1$, $-2^{31} \ldots 2^{31}-1$; целое без знака соответственно в диапазонах $0 \ldots 2^8-1$, $0 \ldots 2^{16}-1$, $0 \ldots 2^{32}-1$; логическую переменную.

Арифметико-логическое устройство (ALU) процессора выполняет операции в соответствии с правилами двоичной арифметики независимо от типа данных. Наряду с самим результатом ALU формирует также признаки нулевого результата ALUZ, отрицательного результата ALUN, переполнения ALUV и переноса ALUC. Признак ALUZ устанавливается в единицу при наличии нулей во всех разрядах результата, признак ALUN — в прямом соответствии с содержимым

старшего разряда результата, признак переполнения — если результат выходит за пределы допустимого диапазона при представлении чисел со знаком. Эта ситуация возникает при $C_n \oplus C_{n-1} = 1$, где C_n — перенос из старшего разряда результата, C_{n-1} — перенос из предыдущего разряда, $n = 7$ (байт), 15 (слово), 31 (длинное слово), $\oplus$ — операция Исключающее ИЛИ. Признаки ALUZ, ALUN, ALUV, ALUC еще не являются кодами условий, а только используются процессором для их установки в зависимости от типа операции. Можно выделить четыре основные группы операций:

1. Типа *сложения*. Признаки ALUN, ALUZ, ALUV, ALUC непосредственно заносятся в соответствующие разряды слова состояния процессора.

2. Типа *вычитания*. В слово состояния процессора заносятся ALUN, ALUZ, ALUV и инверсия ALUC.

3. Типа *сравнения*. В разряд Z заносится содержимое ALUZ N = (ALUN) ∨ (ALUV), V = 0, а в разряд C — инверсия ALUC.

4. *Логическая*. В разряды Z и N заносятся признаки ALUZ и ALUN соответственно, V = 0, содержимое разряда C остается без изменения.

Нетрудно убедиться, что при выполнении операций типа сложения и вычитания разряд V является индикатором переполнения для целых чисел со знаком, разряд C — для целых чисел без знака, а при выполнении операций типа сравнения разряд N является индикатором того, что результат сравнения «меньше» для чисел со знаком, а разряд C — для чисел без знака.

Так как каждый байт, слово или длинное слово могут быть использованы как для представления числа со знаком, так и без знака, необходимо проявлять осторожность при использовании команд перехода вслед за операцией сравнения целых чисел. Рассмотрим два байта $A_1 = 8B$; $A_2 = 3F$ (значения даны в шестнадцатеричном виде). Который из них больше? Это, конечно, зависит от того, представлены в этих байтах восьмиразрядные числа со знаком или без знака. Если числа со знаком, то A_1 отрицательное и, следовательно, A_2 больше. Если это числа без знака, то A_1 больше A_2.

В местах ветвления программы используются инструкции сравнения и проверки (табл. 2.2), которые специально предназначены для установки кодов условий и после которых обычно следуют команды условных переходов. Каждая инструкция может быть использована для проверки одного или сравнения двух элементов данных, представленных в виде целочисленных байтов, слов, двойных слов.

Так как при сравнении двух чисел A_1 и A_2 со знаками признаком того, что A_1 меньше A_2, является отрицательный результат операции $(A_1 - A_2)$, а при сравнении чисел без знака — установка разряда кода условия C, то в СМ1700 имеются два набора инструкций условного перехода, каждый из которых использует соответствующую комбинацию кодов условий.

При использовании беззнаковых переходов к мнемонике инструкции добавляется суффикс U и вместо кода условия N используется C.

Таблица 2.2. Инструкции

Операция	Мнемоника	Условие перехода
Знаковых переходов		
Переход по меньше	BLSS	N = 1
Переход по больше	BGTR	N ∨ Z = 0
Переход по меньше или равно	BLEQ	N ∨ Z = 1
Переход по больше или равно	BGEQ	N = 0
Беззнаковых переходов		
Переход по меньше	BLSSU	C = 1
Переход по больше	BGTRU	C ∨ Z = 0
Переход по меньше или равно	BLEQU	C ∨ Z = 1
Переход по больше или равно	BGEQU	C = 0
Не зависящие от представления данных		
Переход по равенству	BEQ	Z = 1
Переход по неравенству	BNEQ	Z = 0

2.4. ФОРМАТЫ ИНСТРУКЦИЙ

Две особенности архитектуры СМ1700 повышают эффективность программирования. Во-первых, это набор мощных инструкций и режимов адресации, которые позволяют программисту с помощью небольшого числа машинных инструкций создавать сложные конструкции. Во-вторых, способ кодирования инструкций, который обеспечивает эффективное отображение этих команд в оперативной памяти. Формат инструкции ЭВМ СМ1700 показан на рис. 2.1.

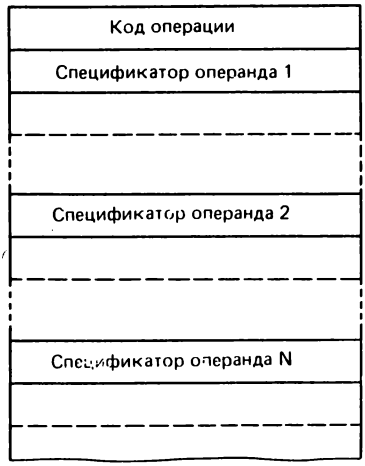


Рис. 2.1. Формат инструкций

В ЭВМ типа СМ4 инструкции кодируются в виде последовательности 16-разрядных слов, и каждое слово делится на ряд полей. Эти поля используются для описания кода операции и спецификаторов операндов. Команды, ориентированные на представление в виде слов, часто требуют больше места, чем это необходимо в действительности, так как они должны занимать целое число слов. Например, рассмотрим представление в памяти инструкции MOV#5,R2 (пересылка константы 5 в регистр R2):

1000	012702
1002	000005

Здесь второе слово занимает константа 5, которая могла бы уместиться в один байт, однако архитектура СМ4 не позволяет этого сделать.

Набор инструкций в СМ1700 спроектирован с учетом обеспечения общности и эффективности представления в оперативной памяти. Инструкции имеют переменную длину и представляются в виде последовательности байтов. Первый байт — это код операции. За кодом операции может следовать до шести спецификаторов операндов, каждый из которых может содержать от одного до девяти байтов, причем содержимое первого байта определяет режим адресации.

Во время выполнения инструкции процессор анализирует первый байт, который содержит код операции. Число последующих спецификаторов операндов определяется кодом операции. Для определения режима адресации процессор анализирует первый байт спецификатора операнда. В зависимости от режима адресации процессору может понадобиться провести анализ различного числа байтов для вычисления эффективного адреса операнда. Первый байт каждого спецификатора операнда делится на две части: в одной определяется режим адресации, а в другой — регистр общего назначения, который используется для адресации.

В процессоре вычислительного комплекса СМ1700 имеются шестнадцать 32-разрядных регистров общего назначения, которые могут использоваться для выполнения арифметических операций, индексации элементов массивов данных и в качестве указателей массивов данных.

Четыре регистра имеют специальное назначение и при обращении к ним программист должен соблюдать осторожность, так как процессор при выполнении некоторых процедур и инструкций автоматически использует их для хранения и модификации различных указателей. Эти регистры имеют специальные названия:

R12 (AP) — указатель аргумента (используется при процедуре вызова подпрограмм).

R13 (FP) — указатель кадра (используется при процедуре вызова подпрограмм).

R14 (SP) — указатель стека (используется для установки и модификации «верхушки» стека).

R15 (PC) — счетчик инструкций (хранит адрес очередной ячейки потока инструкций, обрабатываемого процессором).

Отметим, что регистры R14 и R15 имеют то же назначение, что и регистры R6 и R7 в СМ4.

Остальные регистры R0 — R11 могут быть использованы в спецификаторе операнда без ограничений.

Спецификатор операнда занимает 1 байт. Младшие четыре разряда определяют номер регистра, а старшие — код режима адресации, т. е. способ интерпретации содержимого выбранного регистра.

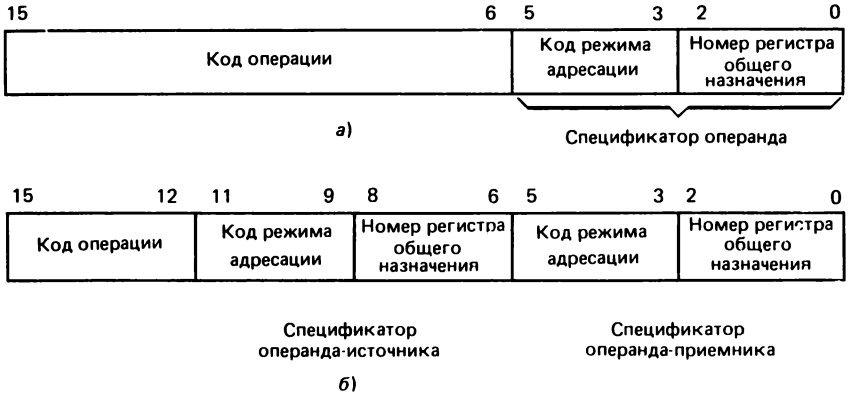


Рис. 2.2. Спецификаторы операнда в режиме совместимости:
а — одноадресные инструкции; *б* — двухадресные инструкции

В режиме совместимости реализуются способы адресации СМ4. Спецификатор операнда содержит шесть разрядов, три из которых определяют номер регистра общего назначения, а остальные — номер режима адресации. В СМ4 имеются восемь 16-разрядных регистров, из которых R6 является указателем стека, а R7 — счетчиком инструкций.

Ниже будут подробно рассмотрены все режимы адресации в СМ1700. Каждый режим иллюстрируется примером. Значения адресов и содержимого ячеек памяти и регистров приводятся в шестнадцатеричном виде. Так как данные имеют различную длину, то их размер определяется числом шестнадцатеричных цифр, использованных для их записи (две цифры — байт, четыре цифры — слово, восемь цифр — длинное слово).

В тех случаях, когда способу адресации соответствует аналогичный способ в режиме совместимости, приводится пример его реализации в СМ4. Так как спецификатор операнда в СМ4 занимает шесть разрядов (три — номер регистра общего назначения и три — режим адресации), то в этих примерах удобнее использовать восьмеричную запись (три цифры — байт, шесть цифр — слово). Расположение спецификаторов операнда в одно- и двухадресных инструкциях в режиме совместимости показано на рис. 2.2, *а*, *б*.

2.5. РЕГИСТРОВЫЙ РЕЖИМ АДРЕСАЦИИ

В этом режиме операндом является содержимое регистра. Рассмотрим инструкцию MOVL R2, R3, которая пересылает содержимое регистра R2 в регистр R3. Код операции для этой инструкции D0. Спецификаторы операндов имеют коды 52 и 53, где цифра 5 является кодом регистрового режима адресации. Пусть код инструкции запоминается по адресу 305:

Представление инструкции в памяти	Содержимое регистров	
00000305: D0	До операции	После операции
00000306: 52	(R2) = 00000010	(R2) = 00000010
00000307: 53	(R3) = 00001000	(R3) = 00000010

В режиме совместимости имеется аналогичный способ адресации с кодом 0. Так как максимальный размер данных в этом режиме — слово, то имеются только две формы мнемоник инструкции. При работе с байтами добавляется суффикс B. В противном случае по умолчанию считается, что инструкция оперирует со словами. Сама инструкция занимает в памяти не менее двух байтов, причем ее начало должно быть всегда выровнено на границу слова (т. е. должно иметь четный адрес). Рассмотрим аналогичную инструкцию MOV R2, R3:

Представление инструкции в памяти	Содержимое регистров	
	До операции	После операции
000306: 010203	(R2) = 000020	(R2) = 000020
	(R3) = 010000	(R3) = 000020

2.6. КОСВЕННО-РЕГИСТРОВЫЙ РЕЖИМ

В этом режиме содержимое регистра является адресом операнда. Мнемоника на ассемблере (R_n). Рассмотрим инструкцию MOVL R2, (R3). Код операции тот же. Спецификатор первого операнда 52, а второго 63, где цифра 6 является кодом косвенно-регистрового режима. При том же содержимом регистров, что и в предыдущем примере, будем иметь:

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: D0	(R2) = 00000010	(R2) = 00000010
00000306: 52	(R3) = 00001000	(R3) = 00001000
00000307: 63	(00001000) = 00000200	(00001000) = 00000010
	Режим совместимости	
000306: 010213	(R2) = 000020	(R2) = 000020
	(R3) = 010000	(R3) = 010000
	(010000) = 001000	(010000) = 000020

Примечание. Код способа адресации равен 1.

2.7. РЕЖИМ С АВТОУВЕЛИЧЕНИЕМ

В этом режиме содержимое выбранного регистра является адресом операнда. После выполнения операции содержимое регистра увеличивается на N. В зависимости от типа операнда:

N = 1, если операндом является байт;
N = 2, если операндом является слово;
N = 4, если операндом является длинное слово;
N = 8, если операндом является учетверенное слово или слово с плавающей запятой;
N = 16, если операндом является целое слово длиной 128 разрядов или двойное слово с плавающей запятой.

Код режима равен 8, код способа адресации в режиме совместимости равен 2. Мнемоника на ассемблере $(R_n)+$. Рассмотрим инструкцию MOVL (R0), (R2)+:

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: D0	(R0) = 00001000	(R0) = 00001000
00000306: 60	(R2) = 00001050	(R2) = 00001052
00000307: 82	(00001000) = 000000AC	(00001000) = 000000AC
	(00001050) = 00000000	(00001050) = 000000AC
	Режим совместимости	
000306: 011022	(R0) = 010000	(R0) = 010000
	(R2) = 010120	(R2) = 010122
	(010000) = 000254	(010000) = 000254
	(010120) = 000000	(010120) = 000254

2.8. РЕЖИМ С АВТОУМЕНЬШЕНИЕМ

В этом режиме содержимое выбранного регистра уменьшается на N, определяемое так же, как и в режиме с автоувеличением. После этого содержимое регистра используется как адрес операнда. Код режима равен 7, код способа адресации в режиме совместимости равен 4. Мнемоника на ассемблере $-(R_n)$. Рассмотрим инструкцию CLRB −(R5). Код операции равен 94:

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 94	(R5) = 00001000	(R5) = 00000FFF
00000306: 75	00000FFF = 1A	00000FFF = 00
	00001000 = 1A	00001000 = 1A
	Режим совместимости	
000306: 005045	(R5) = 010000	(R5) = 007777
	007777 = 032	007777 = 000
	010000 = 032	010000 = 032

2.9. КОСВЕННЫЙ РЕЖИМ С АВТОУВЕЛИЧЕНИЕМ

В этом режиме содержимое выбранного регистра является адресом адреса операнда. Затем содержимое регистра увеличивается на 4 (в режиме совместимости на 2) независимо от размера операнда. Это определяется тем, что регистр содержит адрес адреса (операнда), а адрес всегда занимает длинное слово (в режиме совместимости — слово). Код режима 9 (в режиме совместимости 3). Мнемоника на ассемблере @(R_n)+. Расссмотрим инструкцию CLRB @(R5)+:

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 94	(R5) = 00001000	(R5) = 00001004
00000306: 95	(00001000) = 000000AC	(00001000) = 000000AC
	(000000AC) = 0A	(000000AC) = 00
	Режим совместимости	
000306: 105036	(R5) = 010000	(R5) = 010002
	(010000) = 000254	(010000) = 000254
	(000254) = 012	(000254) = 000

2.10. РЕЖИМ СМЕЩЕНИЯ

В этом режиме содержимое выбранного регистра складывается с содержимым байта, слова или длинного слова, следующего непосредственно за спецификатором операнда. Полученная сумма является адресом операнда. Коды режима: A — смещение представлено байтом; C — смещение представлено словом; E — смещение представлено длинным словом. Мнемоники на ассемблере имеют вид:

B^X(R_n), W^X(R_n), L^X(R_n), где X — смещение.

Возможность выбора размера смещения позволяет экономить память. Так, инструкция CLRW B^2(R1) будет занимать в памяти три байта и представляется в виде:

00000305: B4
00000306: A1
00000307: 02

В режиме совместимости (и, следовательно, в ЭВМ типа СМ4) использовать переменный размер смещения нельзя. Инструкция CLR 2(R1) будет занимать четыре байта и представляется в виде:

000306: 005061
000310: 000002

Примечание. В режиме совместимости код режима адресации равен 6.

В этом режиме содержимое регистра, как правило, используется в качестве базового адреса, некоторой структуры данных, а смещение указывает на конкретный элемент массива, адрес начала которого находится в регистре R_n. Например, очистить третий элемент массива байтов можно с помощью инструкции CLRB B^2(R4), если в R4 находится адрес начала массива:

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 94	(R4) = 00001000	(R4) = 00001000
00000306: A4	(00001000) = 00	(00001000) = 00
00000307: 02	(00001001) = 01	(00001001) = 01
	(00001002) = 02	(00001002) = 00
	Режим совместимости	
000306: 105064	(R4) = 010000	(R4) = 010000
000310: 000002	(010000) = 000	(010000) = 000
	(010001) = 001	(010001) = 001
	(010002) = 002	(010002) = 000

2.11. КОСВЕННЫЙ РЕЖИМ СМЕЩЕНИЯ

В этом режиме содержимое выбранного регистра складывается с содержимым байта, слова или длинного слова, следующего непосредственно за спецификатором операнда. Полученная сумма является адресом длинного слова (в режиме совместимости — слова), которое является адресом операнда. Коды режима: B — смещение представлено байтом; D — смещение представлено словом; F — смещение представлено длинным словом. Мнемоника на ассемблере @B^X(R_n), @W^X(R_n), @L^X(R_n), где X — смещение. Рассмотрим инструкцию MOVW @B^8(R5), (R2):

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: B0	(R5) = 00000100	(R5) = 00000100
00000306: B5	(R2) = 00000400	(R2) = 00000400
00000307: 02	(00000100) = 00000100	(00000100) = 00000100
00000308: 62	(00000104) = 00000200	(00000104) = 00000200
	(00000108) = 00000300	(00000108) = 00000300
	(00000300) = AAAA	(00000300) = AAAA
	(00000400) = 0000	(00000400) = AAAA

000306: 017512	(R5) = 000400	(R5) = 000400
000310: 000010	(R2) = 002000	(R2) = 002000
	(000400) = 000400	(000400) = 000400
	(000402) = 000500	(000402) = 000500
	(000404) = 000600	(000404) = 000600
	(000406) = 000700	(000406) = 000700
	(000410) = 001000	(000410) = 001000
	(001000) = 125252	(001000) = 125252
	(002000) = 000000	(002000) = 125252

Примечание. В режиме совместимости код адресации равен 7.

2.12. РЕЖИМ КОРОТКОГО ЛИТЕРАЛА

Так как многие литералы, используемые в программах, имеют небольшой размер, то в СМ1700 предусмотрен специальный режим адресации, называемый режимом короткого литерала. В этом режиме константа содержится непосредственно в самом спецификаторе операнда. Формат спецификатора короткого литерала показан на рис. 2.3.

Другими словами, любой спецификатор операнда, в котором старшие два разряда равны нулю, содержит литеральную константу в младших шести разрядах. С помощью литерала в инструкции могут быть представлены целые числа без знака в диапазоне от 0 до 63. Мнемоника на ассемблере S^#n, где n — литерал. Рассмотрим инструкцию MOVL S^#18,R3, которая будет представлена в памяти в виде:

```
00000305: D0; код операции
00000306: 18; короткий литерал (18)
00000307; 53; регистровый режим (R3)
```

2.13. ИНДЕКСНЫЙ РЕЖИМ

Одним из наиболее мощных средств адресации в СМ1700 является использование регистров общего назначения для определения индекса элемента массива данных. Формат спецификатора индексного режима показан на рис. 2.4.

Здесь разряды 15...8 содержат второй спецификатор операнда, который называется базовым. Он может определять любой режим адресации, кроме регистрового, короткого литерала и индексного. Если базовый спецификатор требует расширения, то

Рис. 2.3. Спецификатор короткого литерала

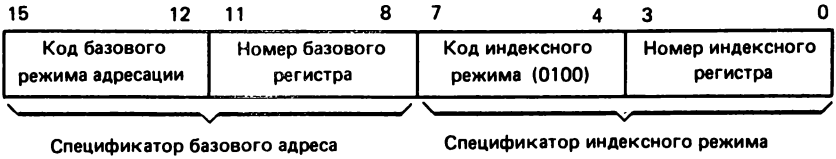


Рис. 2.4. Спецификатор индексного режима

это расширение следует непосредственно за спецификатором. Спецификатор обычно определяет адрес начала массива, а в индексном регистре R_x содержится номер элемента массива. При определении эффективного адреса операнда сначала вычисляется базовый адрес массива или таблицы. Затем содержимое индексного регистра умножается на единицу, двойку, четверку или восьмерку в зависимости от размера операнда и прибавляется к базовому адресу.

Названия вариантов индексного режима формируются из названия режима в базовом спецификаторе с добавлением слова «индексный». Ниже приводятся эти названия и обозначения на ассемблере:

1. Косвенно-регистровый индексный, $(R_n)[R_x]$.
2. С автоувеличением индексный, $(R_n)+[R_x]$.
3. Косвенный с автоувеличением индексный, $@(R_n)+[R_x]$.
4. С автоуменьшением индексный, $-(R_n)[R_x]$.
5. Смещения индексный, $N\hat{}X(R_n)[R_x]$, где N может принимать значения B, W, L.
6. Косвенный смещения индексный, $@N\hat{}X(R_n)[R_x]$.

При работе со структурами данных индексный режим гораздо удобнее, чем просто режим смещения. Во-первых, здесь имеется мощный механизм задания начального адреса массива с помощью одного из основных режимов адресации, что позволяет легко модифицировать этот адрес. Во-вторых, программист указывает в регистре R_x только номер элемента, а необходимое смещение вычисляется автоматически из *контекста* операнда. (Подробнее контекст операнда рассматривается ниже.) Так, инструкция, выполняющая ту же функцию, что и приведенная в 2.10, может выглядеть следующим образом: CLRB (R0)[R4].

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
00000305: 94	(R0)=00001000	(R0)=00001000
00000306: 44	(R4)=00000002	(R4)=00000002
00000307: 60	(00001000)=00	(00001000)=00
	(00001001)=01	(00001001)=01
	(00001002)=02	(00001002)=00

2.14. КОНТЕКСТ ОПЕРАНДА

Каждый операнд в СМ1700 имеет подразумеваемый контекст, определяющий тип элемента данных, над которым производится операция. При использовании наиболее сложных режимов адресации, таких как индексация или автоувеличение, необходимо учитывать контекст операнда для того, чтобы понять, как будет модифицироваться содержимое регистров. Так, контекст операнда (R5)+ определяется его размером: содержимое регистра R5 в инструкциях CLRB и CLR модифицируется различным образом.

Иногда отдельные операнды в одной и той же инструкции имеют разный контекст. Так, в инструкции MOVA, которая используется для пересылки адреса (а не самого операнда) источника по адресу, определяемому операндом-приемником:

1. Первый операнд, или операнд-источник, указывает спецификацию адреса элемента данных. Контекст этого операнда определяется его размером в коде операции и соответственно мнемонике инструкции, например MOVAB, MOVAW или MOWAL для элементов данных в виде байта, слова и двойного слова.

2. Второй операнд, или операнд-приемник, определяет адрес двойного слова, по которому засылается вычисленный 32-разрядный адрес. Контекст этого операнда всегда соответствует двойному слову, так как адрес всегда содержит 32 разряда. Для инструкции MOVAW TEMP, R0 контекст не играет роли, поскольку адрес ячейки TEMP определяется однозначно и над регистром не производится никаких дополнительных операций, например автоувеличения. Адрес ячейки TEMP не зависит от ее собственной длины. Поэтому независимо от того, выполняется ли инструкция MOVAL или MOVAB, результат остается тем же самым. Программист может выбирать любую из форм представления инструкции, руководствуясь соображениями повышения наглядности записи и подчеркивая размер операнда.

В отличие от приведенного выше примера в инструкции MOVAW (R0)+,(R1)+ контекст играет большую роль. При исполнении этой инструкции адрес слова, на который указывает содержимое регистра R0, пересылается в двойное слово, на которое указывает содержимое регистра R1. В результате выполнения инструкции содержимое регистра R0 будет увеличено на два, а содержимое регистра R1 на четыре.

2.15. РЕЖИМЫ АДРЕСАЦИИ С ИСПОЛЬЗОВАНИЕМ СЧЕТЧИКА ИНСТРУКЦИЙ

Одной из основных особенностей рассмотренных выше способов адресации является то, что во всех них требуется предварительная загрузка одного из регистров общего назначения.

После того, как регистр загружен, его содержимое может быть использовано в качестве указателя текущего адреса массива, базового адреса начала массива, индексного смещения к конкретному элементу массива и т. п. При однократном обращении к произвольной ячейке памяти такой способ оказывается неудобным, так как требует выполнения сразу двух операций: предварительной загрузки адреса ячейки в регистр; собственно обращения к ячейке через регистр с помощью одного из режимов адресации. В действительности эту операцию можно реализовать гораздо проще, если учесть, что счетчик инструкций R15 (или R7 в режиме совместимости) является регистром общего назначения и может быть использован в любом из режимов адресации. Однако не для всех режимов адресации это имеет смысл. Так, если использовать счетчик инструкций в качестве регистра в режиме с автоуменьшением, то это приведет к непредсказуемому результату. Фактически со счетчиком инструкций используются только четыре режима адресации: с автоувеличением, косвенный с автоувеличением, смещения и косвенный смещения. Это дает (с точки зрения программиста, но не аппаратно) четыре дополнительных режима адресации: непосредственный, абсолютный, относительный и косвенно-относительный.

Рассмотрим алгоритм выполнения режима с автоувеличением при использовании счетчика инструкций в качестве регистра общего назначения. По определению режима операция производится над операндом, адрес которого находится в выбранном регистре. Если этим регистром является счетчик инструкций, то он в этот момент будет указывать на ячейку, непосредственно следующую за спецификатором операнда. Таким образом, операнд оказывается непосредственно в потоке инструкций. После выборки операнда (по определению режима) содержимое счетчика инструкций увеличится на размер операнда, определяемого кодом операции. Поэтому длина константы должна соответствовать типу инструкции, даже если ее фактический размер меньше. В режиме совместимости непосредственный операнд занимает слово (даже если его длина не превышает длину байта). Поэтому счетчик инструкций всегда увеличивается на 2. Рассмотрим инструкцию MOVB #10,(R2).

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 90	(R2) = 00001000	(R2) = 00001000
00000306: 8F	(PC) = 00000305	(PC) = 00000309
00000307: 10	(00001000) = 00	(00001000) = 10
00000308: 62		

Режим совместимости

000306: 112712	(R2) = 010000	(R2) = 010000
000310: 000020	(PC) = 000306	(PC) = 000312
	(010000) = 00	(010000) = 20

Если использовать косвенный режим с автоувеличением, то по определению режима содержимое счетчика инструкций является не адресом операнда, а адресом адреса, а затем это содержимое увеличивается на 4 (т. е. на значение длинного слова). Таким образом, мы получаем возможность задать непосредственно в инструкции абсолютный адрес операнда. Если в предыдущем примере использовать этот режим, то по адресу 00001000 будет занесено не число 10, а содержимое ячейки 10. На ассемблере такая инструкция будет иметь мнемонику MOVB @#10, (R2):

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 90	(R2) = 00001000	(R2) = 00001000
00000306: 9F	(PC) = 00000305	(PC) = 0000030C
00000307: 10	00000010 = 0A	00000010 = 0A
00000308: 00	00001000 = 00	00001000 = 0A
00000309: 00		
0000030A: 00		
0000030B: 62		
	Режим совместимости	
000306: 013712	(R2) = 010000	(R2) = 010000
000310: 000020	(PC) = 000306	(PC) = 000312
	000020 = 12	000020 = 12
	010000 = 00	010000 = 12

Есть определенный класс программ, которые называются позиционно-независимыми. Они могут быть загружены и запущены в любой области памяти без перетрансляции или перекомпоновки.

При перемещении программы в памяти адрес ячейки, заданной косвенно-регистровым режимом, остается неизменным. Это неудобно при адресации общих ресурсов системы, к которым разрешен доступ программе. Если адресуемые данные перемещаются вместе с программой, то использование этого режима адресации приведет к неверному результату. Что сделать в этом случае? Рассмотрим, как использовать счетчик инструкций в режиме со смещением.

Содержимое выбранного регистра (в нашем случае счетчика инструкций) складывается со смещением, которое имеется непосредственно в потоке инструкций. Полученная сумма является адресом операнда. Очевидно, что если смещение подобрать таким образом, чтобы оно было равно разнице между текущим адресом инструкции (точнее, самого кода смещения) и адресом операнда,

то сумма будет равна адресу операнда. Теперь при любом перемещении программы операнд будет адресоваться верно.

Этот режим называется относительным и на ассемблере обозначается просто A, где A — адрес операнда. Рассмотрим инструкцию MOVB 10, (R2):

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 90	(R2) = 00001000	(R2) = 00001000
00000306: CF	(PC) = 00000305	(PC) = 00000310
00000307: 07	00000010 = 0A	00000010 = 0A
00000308: FD	00001000 = 00	00001000 = 0A
00000309: 62		
	Режим совместимости	
000306: 116712	(R2) = 010000	(R2) = 010000
000310: 117506	(PC) = 000306	(PC) = 000312
	000020 = 012	000020 = 012
	010000 = 000	010000 = 012

Отметим, что смещение в данном примере отрицательно и представляется в виде слова в дополнительном коде. Длина смещения (байт, слово, длинное слово) и соответственно конкретный код режима определяются автоматически на этапе ассемблирования.

И, наконец, если использовать счетчик инструкций в косвенном режиме со смещением, то получим еще один способ адресации, который называется косвенно-относительным. Здесь сумма содержимого счетчика инструкций и смещения является адресом адреса операнда. Для программиста этот способ является просто косвенной адресацией с использованием произвольной ячейки для хранения адреса операнда. Как и в предыдущем режиме, смещение должно соответствовать разнице между адресом этой ячейки и текущим положением инструкции. На ассемблере этот режим обозначается @A, где A — адрес ячейки, в которой хранится адрес операнда. Рассмотрим инструкцию MOVB @10, (R2):

Представление инструкции в памяти	Содержимое регистров и ячеек памяти	
	До операции	После операции
	Собственный режим	
00000305: 90	(R2) = 00001000	(R2) = 00001000
00000306: CC	(PC) = 0000305	(PC) = 00000310
00000307: 07	00000010 = 0000000A	00000010 = 0000000A
00000308: FD	0000000A = 5D	0000000A = 5D
00000309: 62	00001000 = 00	00001000 = 5D

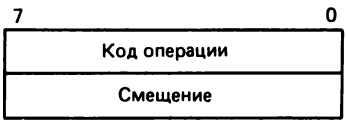


Рис. 2.5. Формат инструкций перехода

2.16. АДРЕСАЦИЯ ПЕРЕХОДОВ

Последний вид адресации, который мы будем рассматривать, это адресация переходов. Формат инструкции условного перехода показан на рис. 2.5.

Адресация в этих инструкциях отличается от всех рассматриваемых ранее тем, что здесь нет спецификатора операнда. Каждая инструкция условного перехода состоит из кода операции и следующего за ним байтного смещения со знаком. При формировании адреса перехода смещение со знаком складывается с содержимым счетчика команд подобно тому, как это делалось при относительном режиме адресации. Такой способ обусловлен тем, что большинство переходов адресованы к ячейкам, находящимся на небольшом расстоянии от самой инструкции перехода (в пределах ± 128 байт). Лишь в некоторых случаях требуется использование более эффективных режимов адресации.

2.17. ИСПОЛЬЗОВАНИЕ РЕЖИМОВ АДРЕСАЦИИ

Рассмотренные выше режимы адресации позволяют эффективно программировать различные задачи. В этом разделе будут приведены примеры их использования для реализации задачи формирования массива данных, элементы которого представляют собой сумму элементов двух других массивов. На языке высокого уровня Фортран программа для решения такой задачи будет выглядеть следующим образом:

```
      DO   1  I=1,N
      C(I) = A(I) + B(I)
1     CONTINUE
```

где N — число элементов массива.

Ниже мы рассмотрим несколько способов реализации этой программы на ассемблере, постепенно приближаясь к наиболее эффективному решению.

Каждый из трех массивов, используемых в нашем примере, на ассемблере состоит из 11 32-разрядных целочисленных элементов. Начало каждого из массивов помечено метками A B и C соответственно. Сначала покажем простое линейное решение задачи. Для адресации произвольной ячейки используется режим смещения со счетчиком инструкций. Для программиста этот режим можно назвать *простым*, так как в этом случае он просто указывает символьный адрес требуемой ячейки, например INCW AB (увеличить на единицу содержимое ячейки AB). Ассемблер сам определяет, что в этом случае нужно использовать режим смещения со счетчиком инструкций, и вычисляет требуемое смещение.

Программа на ассемблере будет иметь вид:

```
ADDL3   A,B,C           ;Сформировать 1-й элемент массива C
ADDL3   A+4,B+4,C+4     ;Сформировать 2-й элемент массива C
ADDL3   A+8,B+8,C+8     ;Сформировать 3-й элемент массива C
ADDL3   A+N,B+N,C+N     ;Сформировать N-й элемент массива C
```

Очевидно, что использование простого режима при работе с массивами данных является неэффективным. Можно добиться существенного повышения быстродействия и уменьшить объем занимаемой памяти, если для адресации элементов массива использовать регистры общего назначения в режиме с автоувеличением:

```
A:   BLKL    N              ;Зарезервировать место для
                            ;элементов массива A
B:   BLKL    N              ;Зарезервировать место для
                            ;элементов массива B
C:   BLKL    N              ;Зарезервировать место для
                            ;элементов массива C
     MOVL    A,R0           ;Загрузить адрес массива A в R0
     MOVL    B,R1           ;Загрузить адрес массива B в R1
     MOVL    C,R2           ;Загрузить адрес массива C в R2
     ADDL3   (R0)+,(R1)+,(R2)+ ;Сформировать 1-й элемент массива C
               .
               .
               .
   ADDL3     (R0)+,(R1)+,(R2)+ ;Сформировать N-й элемент массива C
```

В начале программы используется непосредственный режим для загрузки адресов начала массивов в соответствующие регистры. Так как адрес ячейки всегда занимает длинное слово, то в инструкциях такого типа нужно добавлять суффикс L, который определяет контекст операнда как длинное слово. В противном случае инструкция будет работать неверно, даже если фактическая длина адреса меньше длинного слова. Эту инструкцию в соответствии с режимом адресации можно представить в виде

MOVL (PC)+,R0

Таким образом, содержимое счетчика инструкций будет увеличиваться на размер операнда, определяемый контекстом.

Для того чтобы избавить программиста от необходимости помнить о контексте операнда, в СМ1700 предусмотрена специальная инструкция засылки адреса, в которой суффикс определяет контекст только операнда-приемника, а операнд-источник (т. е. пересылаемый адрес) всегда считается длинным словом. С использованием этой инструкции начало программы будет выглядеть следующим образом:

```
MOVAL   A,R0    ;Переслать адрес начала массива A в R0
MOVAL   B,R1    ;Переслать адрес начала массива B в R1
MOVAL   C,R2    ;Загрузить адрес начала массива C в R2
```

При большом числе элементов массива программу следует оформить в виде цикла:

```
A:      BLKL    N           ;Зарезервировать место для массива A
B:      BLKL    N           ;Зарезервировать место для массива B
C:      BLKL    N           ;Зарезервировать место для массива C
        MOVAL   A,R0        ;Переслать адрес начала массива A в R0
        MOVAL   B,R1        ;Переслать адрес начала массива B в R1
        MOVAL   C,R2        ;Загрузить адрес начала массива C в R2
        MOVL    #N,R3       ;Установить счетчик числа итераций
        ADDL3   (R0)+,(R1)+,(R2)+ ;Сформировать i-й элемент массива C
        DECL    R3          ;Уменьшить на 1 счетчик итераций
        BNE     M           ;Переход по адресу M,пока счетчик итераций
                            ;не равен 0
```

Здесь снова использовался непосредственный режим адресации для задания числа итераций цикла. Если это число умещается в шести двоичных разрядах, т. е. $0 \leqslant N \leqslant 63$ или $-32 \leqslant 11 \leqslant 31$, то ассемблер автоматически выберет режим короткого литерала.

И, наконец, можно еще больше сократить текст программы, если воспользоваться индексным режимом адресации:

```
A:      BLKL    N               ;Зарезервировать место для массива A
B:      BLKL    N               ;Зарезервировать место для массива B
C:      BLKL    N               ;Зарезервировать место для массива C
        CLRL    R0              ;Очистить регистр R0
        MOVL    #N,R3           ;Установить счетчик числа итераций
        ADDL3   A[R0],B[R0],C[R0] ;Сформировать i-й элемент массива C
        INCL    R0              ;Увеличить на 1 номер элемента
        DECL    R3              ;Уменьшить на 1 счетчик итераций
        BNEQ    M               ;Переход по адресу M, если не равно 0
```

Здесь содержимое регистра фактически является номером элемента. Так как мы производим операции с элементами массивов A, B и C, которые имеют одинаковые номера, то нам требуется только один регистр.

В этой главе были подробно рассмотрены способы адресации операндов, а также отображение инструкций в памяти. Приведем сводные таблицы режимов адресации, в которых даны названия режимов, их коды и краткое описание (табл. 2.3—2.5).

Таблица 2.3. **Общие режимы адресации СМ1700**

Наименование режима	Ассемблерная нотация	Код	Описание
Короткий литерал	S^#литерал	0—3	Значение литерала содержится в разрядах с 0-го по 5-й спецификатора операнда
Индексный	база [R_x]	4	Адрес операнда формируется следующим образом: сначала вычисляется адрес начала массива, затем этот адрес складывается с содержимым регистра, умноженным на размер элемента массива в байтах
Регистровый	R_n	5	Регистр содержит операнд
Косвенно-регистровый	(R_n)	6	Регистр содержит адрес операнда

Продолжение табл. 2.3

Наименование режима	Ассемблерная нотация	Код	Описание
Автоувеличение	$(R_n)+$	8	Регистр используется в качестве адреса операнда, затем его содержимое увеличивается на размер операнда в байтах
Автоуменьшение	$-(R_n)$	7	Содержимое регистра уменьшается на размер операнда в байтах, затем используется в качестве адреса операнда
Косвенный с автоувеличением	$@(R_n)+$	9	Содержимое регистра используется в качестве адреса адреса операнда, а затем увеличивается на 4
Байтовое смещение	$B\^X(R_n)$	A	Знак смещения расширяется в старшие разряды, затем смещение складывается с содержимым регистра, полученная сумма является адресом операнда
Косвенное байтовое смещение	$@B\^X(R_n)$	B	Знак смещения расширяется в старшие разряды, затем смещение складывается с содержимым регистра, полученная сумма является адресом адреса операнда
Смещение в словах	$W\^X(R_n)$	C	Знак смещения расширяется в старшие разряды, затем складывается с содержимым регистра, полученная сумма является адресом операнда
Косвенное смещение в словах	$@W\^X(R_n)$	D	Знак смещения расширяется в старшие разряды, затем смещение складывается с содержимым регистра, полученная сумма является адресом адреса операнда
Смещение в двойных словах	$L\^X(R_n)$	E	Смещение складывается с содержимым регистра, полученная сумма является адресом операнда
Косвенное смещение в двойных словах	$@L\^X(R_n)$	F	Смещение складывается с содержимым регистра, полученная сумма является адресом адреса операнда

Таблица 2.4. **Режимы адресации с использованием счетчика команд**

Наименование режима	Ассемблерная нотация	Код	Описание
Непосредственный	# константа	8	Непосредственный операнд находится в потоке инструкций вслед за спецификатором. Ассемблируется в виде режима адресации (PC)+
Абсолютный	@#адрес	9	Адрес операнда находится в потоке инструкций вслед за спецификатором. Ассемблируется в виде режима адресации @(PC)+
Относительный	Адрес	A, B, C	Адрес операнда формируется сложением содержимого счетчика инструкций со смещением (байт, слово или двойное слово), следующим за спецификатором операнда. Ассемблируется в виде режима адресации X(PC

Окончание табл. 2.4.

Наименование режима	Ассемблерная нотация	Код	Описание
Косвенно-относительный	Адрес	D, E, F	Адрес адреса операнда формируется сложением содержимого счетчика инструкций со смещением в байтах, словах или двойных словах, которое следует за спецификатором операнда. Ассемблируется в виде режима адресации @X(PC)

Таблица 2.5. **Индексные режимы адресации**

Наименование режима	Ассемблерная нотация	Описание
Косвенно-регистровый с индексацией	$(R_n)[R_x]$	Индексное смещение формируется умножением содержимого регистра на размер операнда в байтах. Адрес операнда формируется сложением содержимого R_n и индексного смещения
Индексный с автоувеличением	$(R_n)+[R_x]$	Совпадает с предыдущим режимом, за исключением того, что после вычисления адреса операнда содержимое R_n инкрементируется
Индексный с автоуменьшением	$-(R_n)[R_x]$	Совпадает с косвенно-регистровым, за исключением того, что содержимое R_n декрементируется, после чего вычисляется адрес операнда
Относительный с индексацией	$X(R_n)[R_x]$	Базовый адрес формируется сложением содержимого регистра со смещением в байтах, словах или двойных словах. Адрес операнда вычисляется в виде суммы базового адреса и соответствующим образом сформированного индексного смещения
Косвенно-относительный с индексацией	$@X(R_n)[R_x]$	Адрес базового адреса формируется сложением содержимого регистра со смещением. Затем производится выборка базового адреса, который складывается с соответствующим образом сформированным индексным смещением. Полученный результат является адресом операнда
Косвенный с автоувеличением и индексацией	$@(R_n)+[R_x]$	R_n содержит адрес базового адреса, который выбирается и складывается с индексным смещением; R_n затем увеличивается на 4

Глава 3. ОРГАНИЗАЦИЯ ЦИКЛОВ И ПОДПРОГРАММ

3.1. ЦИКЛЫ И СЛОЖНЫЕ ПЕРЕХОДЫ

Простые инструкции переходов были рассмотрены в гл. 2. Были даны два вида условных переходов (знаковый и беззнаковый) и два безусловных перехода BRB и BRW, которые выполняют переход к ячейкам памяти внутри 8- или 16-разрядного смещения соответственно. Однако они не позволяют использовать сложные режимы адресации и не способны передавать управление за пределы 64 Кбайт. Для удовлетворения таких требований существует инструкция JMP — «Универсальный переход». В отличие от инструкций переходов BRB и BRW, в которых инструкция-приемник определяется смещением, прибавленным или вычтенным из регистра PC, приемник инструкции JMP может быть определен любым из допустимых режимов адресации, кроме регистрового, и может быть любой ячейкой памяти внутри адресного пространства программы пользователя. Ниже приведены несколько примеров использования инструкции JMP:

```
JMP     CONT      ;Переход к ячейке CONT
JMP     @ADDR     ;Ячейка ADDR содержит адрес цели
JMP     (R1)      ;Регистр R1 содержит адрес цели
JMP     @(R2)+    ;Регистр R2 указывает на двойное слово, содержащее
                  ;адрес цели. Увеличить R2 на 4 и передать управление
```

Обычно использование возможности универсальной адресации инструкции JMP возникает тогда, когда требуется выполнить переход на основе числового кода. Одним из примеров использования инструкции JMP может быть программный имитатор калькулятора, в котором передача управления выполняется к функциям «Синус», «Косинус», «Тангенс» и «Котангенс» в зависимости от значения переменной IND (от 0 до 3 соответственно). Если адреса этих функций помещены в таблице TBL, то может быть использован косвенный переход, чтобы передать управление соответствующей арифметической функции:

```
TBL:    .ADRESS   SIN,COS,TG,CTG ;Адреса функций
CALCUL:
        MOVL      IND,R2         ;Загрузить код функции
        MOVL      TBL[R2],R2     ;Получить адрес функции
        JMP       (R2)           ;Переход к функциям
RETURN:                          ;Возврат сюда после выполнения функции
```

Этот пример еще раз иллюстрирует использование индексации в СМ1700. Например, если значение переменной IND равно 3, то вторая инструкция MOVL загрузит в R2 двойное слово TBL+12 (адрес функции «Котангенс»), а инструкция JMP затем передаст управление этой функции.

Задача перехода к программной функции, зависящей от значения какой-то переменной, встречается достаточно часто,

поскольку некоторые языки высокого уровня содержат специальные конструкции для обработки таких операторов, как вычисляемый GO TO в языке Фортран или CASE в языке Паскаль. Поэтому набор инструкций в CM1700 включает инструкцию CASE (выбор) для эффективного программирования таких управляющих структур. Инструкция CASE выполняет не только передачу управления, но и инициализацию и проверку границ для индексируемой переменной. Общая форма инструкции CASE приведена ниже:

```
        CASE            SELECTOR,BASE,DISP
TBL:    СМЕЩЕНИЕ_0
        СМЕЩЕНИЕ_1
        .
        .
        СМЕЩЕНИЕ_N-1
NEXT:
```

Целью инструкции CASE является передача управления к одной из N ячеек, базируемых на значении целочисленного операнда SELECTOR. Операнд BASE определяет базовую (нижнюю) границу для операнда SELECTOR. Непосредственно за инструкцией CASE располагается таблица однословных смещений для N ячеек перехода.

Содержимое операнда BASE вычитается из операнда SELECTOR, и полученная разница (индекс) сравнивается с операндом DISP. Если индекс меньше или равен операнду DISP (в беззнаковом диапазоне), то к счетчику инструкций PC прибавляется смещение, выбираемое из таблицы выбора TBL по значению индекса. Другими словами, если значение операнда SELECTOR находится между значениями BASE и BASE+DISP, то значение SELECTOR — BASE используется для выбора элемента таблицы, по которому вычисляется адрес перехода к инструкции-приемнику. Если значение операнда SELECTOR не находится между BASE и BASE+DISP, то управление передается на инструкцию, следующую непосредственно за таблицей выбора.

Важно отметить, что используются смещения, а не фактические адреса. Таким образом, чтобы сформировать смещение, вычисляется и запоминается в таблице разность между адресом перехода и базовым адресом таблицы выбора, вследствие чего инструкция CASE является позиционно-независимой.

Существуют три формы инструкции: CASEB, CASEW и CASEL. Они определяют тип данных операндов. Заметим, что если в мнемонике инструкции отсутствует буква, определяющая тип обрабатываемых данных, то по умолчанию имеется в виду W. Это замечание относится ко всем инструкциям, имеющим модификации по типам данных.

Простым примером использования инструкции CASE является вычисляемый оператор GO TO в языке Фортран, где передача управления выполняется к одному из помеченных операторов, базируемому на значении переменной ITEM:

```
GO TO (10, 20, 30, 40, 50),ITEM
```

Этот же пример на языке Паскаль с помощью оператора CASE можно представить в следующем виде:

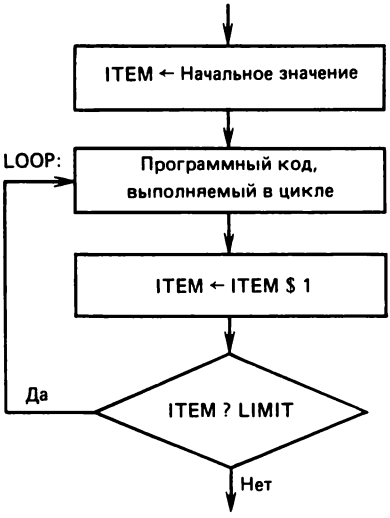
```
CASE    ITEM    OF
 1:         <ОПЕРАТОР ДЛЯ ITEM=1>;
 2:         <ОПЕРАТОР ДЛЯ ITEM=2>;
            .
            .
 5:         <ОПЕРАТОР ДЛЯ ITEM=5>;
```

И соответственно на языке ассемблера СМ1700:

```
        CASEL   ITEM,#1,#4      ;Для ITEM от 1 до 5
CASETBL:
        .WORD   L100-CASETBL    ;Если ITEM=1, переход на L100
        .WORD   L200-CASETBL    ;Если ITEM=2, переход на L200
        .WORD   L300-CASETBL    ;Если ITEM=3, переход на L300
        .WORD   L400-CASETBL    ;Если ITEM=4, переход на L400
        .WORD   L500-CASETBL    ;Если ITEM=5, переход на L500
        BRW     ERROR           ;Если ITEM > 5 или < 1, то
                                ;переход на ERROR
```

Для эффективной организации циклов предназначены инструкции SOB («Вычесть единицу и перейти») и AOB («Прибавить единицу и перейти»). Существуют две модификации каждой инструкции:

```
SOBGEQ  ITEM,LOOP       ;Для ITEM от начального значения
                        ;до конечного (ноль)
SOBGTR  ITEM,LOOP       ;Для ITEM от начального значения
                        ;до конечного (ноль)
AOBLEQ  LIMIT,ITEM,LOOP ;Для ITEM от начального значения
                        ;до конечного (LIMIT)
AOBLSS  LIMIT,ITEM,LOOP ;Для ITEM от начального значения
                        ;до конечного (LIMIT-1)
```



Операция $ модификации счетчика:

\+ для AOBLEQ и AOBLSS
– для SOBGEQ и SOBGTR

Сравнение счетчика с пределом:

? { < для AOBLSS; ⩽ для AOBLEQ; ⩾ для SOBGEQ; > для SOBGTR

для SOBGEQ и SOBGTR предел равен нулю

Рис. 3.1. Организация оператора цикла с использованием инструкций AOB и SOB

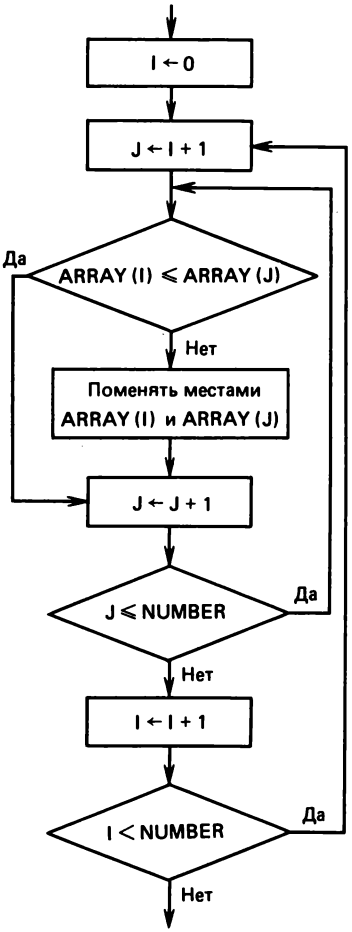


Рис. 3.2. Блок-схема программы сортировки

В этих инструкциях ITEM — счетчик циклов, LIMIT — конечное значение счетчика, LOOP — адрес перехода для продолжения цикла.

Общая форма организации циклов с использованием инструкций SOB и AOB приведена на рис. 3.1.

Использование инструкции AOB иллюстрируется на примере сортировки массива, где меньшие значения перемещаются к верху массива. Если ARRAY — массив элементов, которые должны быть отсортированы по значению, а NUMBER — число элементов в массиве, то алгоритм сортировки на языке Паскаль может быть выражен так:

```
FOR I:=0 TO NUMBER-1 DO
BEGIN
        FOR J:=I+1 TO NUMBER DO
        IF ARRAY[I] < ARRAY[J] THEN
        BEGIN
                TEMP:=ARRAY[I];
                ARRAY[I]:=ARRAY[J];
                ARRAY[J]:=TEMP;
        END;
END;
```

Блок-схема алгоритма сортировки показана на рис. 3.2, программа на ассемблере приведена ниже:

```
        MOVL    NUMBER,R1           ;Получить число элементов
        DECL    R1                  ;Индекс последнего элемента
        MOVAL   ARRAY,R2            ;Базовый адрес массива
        CLRL    R3                  ;Инициализировать I
LOOP:   ADDL3   #1,R3,R4            ;Инициализировать J к I+1
INLOOP: CMPL    (R2)[R3],(R2)[R4]   ;Проверка упорядоченности
        BLEQ    TEST                ;Переход, если первый элемент
                                    ;меньше или равен второму
        MOVL    (R2)[R3],TEMP       ;Сохранить ARRAY(I)
        MOVL    (R2)[R4],(R2)[R3]   ;Поменять ARRAY(I)
        MOVL    TEMP,(R2)[R4]       ;и ARRAY(J)
TEST:   AOBLEQ  R1,R4,INLOOP        ;Продвинуть J
        AOBLSS  R1,R3,LOOP          ;Продвинуть I
```

Полезной является обобщенная инструкция ACB («Сложение, сравнение и переход»), которая эффективно реализует общие

циклы типа FOR или DO, имеющиеся во многих языках высокого уровня. Инструкция ACB позволяет конструировать циклы с увеличением или уменьшением значений индекса.

Выполнение инструкции ACB LIMIT,STEP,INDEX,LOOP заключается в том, что значение приращения STEP добавляется к значению индекса INDEX, который сравнивается с предельным значением LIMIT, и если индекс не достиг предела, то выполняется переход по адресу LOOP. Если приращение отрицательное, то цикл повторяется, пока INDEX $\geqslant$ LIMIT, а если приращение положительное, то цикл повторяется до тех пор, пока INDEX $\leqslant$ LIMIT.

Инструкция ACB использует слово смещения для адресации перехода и может быть использована с операндами LIMIT, STEP и INDEX каждый длиной в байт, слово, двойное слово или в формате с плавающей запятой одинарной и двойной точности. Например, чтобы реализовать фортрановский цикл

```
        DO 30 CLE=10,0,-2
          .
          .
30      CONTINUE
```

который повторяется 6 раз, пока переменная CLE принимает значения 10, 8, ..., 0, может быть использована инструкция ACB:

```
        MOVL    #10,CLE                 ;Инициализировать CLE
DO_CONT:
          .
          .
        ACBL    #0,#-2,CLE,DO_CONT      ;Увеличить и проверить CLE
```

3.2. ИСПОЛЬЗОВАНИЕ СТЕКА

Стек — это зона памяти (т. е. массив непрерывных ячеек данных), используемая для запоминания промежуточных результатов и информации вызова подпрограмм и процедур. Элементы данных динамически добавляются в стек или извлекаются из него по правилу «последним вошел — первым вышел», т. е. элемент, извлекаемый из стека, это всегда элемент, который был последним загружен в стек. Переменная, называемая «указатель стека», всегда указывает на последний элемент, помещенный в стек. Этот последний элемент называется «вершина стека».

В СМ1700 любой общий регистр может использоваться как указатель стека. Принято, что стеки растут в отрицательном направлении, т. е. в сторону младших адресов памяти. Чтобы поместить новый элемент в стек, используется режим адресации с автоуменьшением (вычисляется размер элемента из указателя стека и элемент пересылается в память, отведенную для стека). Например, инструкция MOVL ITEM,–(R6) продвигает указатель стека к следующему двойному слову (из R6 вычитается 4) и помещает сюда значение переменной ITEM.

При удалении из стека используется режим адресации с автоувеличением. Выполняя инструкцию MOVL (R6)+,R8 верхний элемент стека пересылается в R8, а указатель стека возвращается к предыдущей вершине стека (к R6 прибавляется 4).

В СМ1700 регистр R14 закреплен за указателем стека и обычно обозначается SP (Stack Pointer — указатель стека) на языке ассемблера. При загрузке программы операционная система автоматически отводит блок памяти в области пользователя и загружает в SP адрес блока.

Некоторые инструкции по умолчанию используют SP для обращения к стеку. Например, инструкция PUSHL R5 загружает содержимое R5 в стек по указателю SP. Эта инструкция эквивалентна инструкции MOVL R5,−(SP), занимает меньше места в памяти, причем выполняется быстрее. Подобным образом можно написать POPL R5, чтобы извлечь верхний элемент из стека и загрузить его в R5 (В СМ1700 в действительности нет инструкции POPL, поскольку, как будет видно в дальнейшем, существует специальный аппарат для автоматического удаления элементов из стека. Однако ассемблер распознает «POPL TEMP» и вместо нее генерирует «MOVL (SP)+,TEMP».)

В ранее рассмотренной программе сортировки вместо использования переменной TEMP для хранения ARRAY(I) в процессе обмена лучше было бы использовать стек:

```
PUSHL   (R2)[R3]              ;Сохранить ARRAY(I) в стеке
MOVL    (R2)[R4],(R2)[R3]     ;Заменить ARRAY(I)
POPL    (R2)[R4]              ;Заменить ARRAY(J)
```

Помимо повышения скорости выполнения это позволяет более эффективно использовать оперативную память, поскольку зону стека может использовать любая другая программа.

Стек часто используется для сохранения и восстановления общих регистров. В приведенном ниже фрагменте программы регистры 5—7 сохраняются и затем восстанавливаются через стек (заметим, что регистры восстанавливаются в обратном порядке):

```
PUSHL   R7              ;Сохранить R7
PUSHL   R6              ;Сохранить R6
PUSHL   R5              ;Сохранить R5
  .
  .
  . <инструкции, которые модифицируют R5 - R7>
  .
POPL    R5              ;Восстановить R5
POPL    R6              ;Восстановить R6
POPL    R7              ;Восстановить R7
```

Поскольку в программах регистры сохраняются и восстанавливаются довольно часто, то в СМ1700 имеются специальные инструкции PUSHR (загрузка общих регистров в стек) и POPR (извлечение общих регистров из стека), предназначенные для запоминания и восстановления сразу нескольких регистров. Опе-

рандом для этих инструкций является 16-разрядная маска, где каждому единичному биту соответствует определенный регистр. Ассемблерный синтаксис ^M<R_a, R_b, ..., R_n> автоматически генерирует маску с соответствующим набором битов для запоминаемых или восстанавливаемых регистров. Например, предыдущий фрагмент может быть заменен следующей последовательностью:

```
PUSHR   #^M<R5,R6,R7>   ;Сохранить три регистра
  .
  . <инструкции, которые модифицируют R5 - R7>
  .
  .
PUSHR   #^M<R5,R6,R7>   ;Сохранить три регистра
```

Инструкция PUSHR загружает в стек регистры в порядке от старшего к младшему, в то время как инструкция POPR восстанавливает их в порядке от младшего к старшему независимо от того, как они указаны в маске.

Стек часто используется для того, чтобы содержать аргументы вызванной подпрограммы. Поскольку адреса, как и значения, могут передаваться подпрограммам, в СМ1700 имеются инструкции типа PUSHA (загрузка адреса в стек) для запоминания в стеке адресов. Например, инструкция PUSHAL ITEM загружает адрес переменной ITEM в стек так же, как MOVAL ITEM,-(SP), но быстрее и компактнее.

Существуют четыре модификации инструкции загрузки адреса в стек: PUSHAB, PUSHAW, PUSHAL и PUSHAQ. Каждая инструкция загружает 32-разрядный адрес в стек. Указание на тип данных используется при вычислении контекстно-зависимых адресов; например в инструкции PUSHAW ARRAY[R3] указание контекста необходимо для вычисления фактического адреса.

Данный параграф завершает программа перекодировки, в которой используется стек. В этой программе (рис. 3.3) берется целое двоичное число SIZE, перекодируется в десятичное символьное представление и используется макрокоманда OUTCHAR для вывода на видеотерминал числа в 8-битовом коде СМ1700 (см. приложение 3). Стек используется потому, что алгоритм формирует десятичные цифры от наименее значащей к наиболее значащей. На терминал они должны быть выведены в обратном порядке. Десятичная цифра берется в двоичном коде и ее эквивалент в 8-битовом коде СМ1700 вычисляется добавлением к ней кода нуля (30 в коде СМ1700).

В этом примере вводится новая инструкция EDIV (расширенное деление), которая делит 64-разрядное делимое на 32-разрядный делитель, вырабатывая 32-разрядное частное и 32-разрядный остаток. Формат инструкции EDIV следующий:

EDIV делитель, делимое, частное, остаток

В рассмотренном примере старшая половина четырехсловного делимого обнуляется перед началом цикла. Следует заметить, что инструкция CLRQ, когда указан регистровый операнд,

```
        ;Вывод целого числа в 8-битовом коде СМ1700
        ;
        ;Вход:
        ;
        ;       SIZE содержит вычисляемую величину
        ;
        ;Выход:
        ;
        ;       Вывод значения в коде СМ1700 посредством
        ;       макроса OUTCHAR
        ;
        ;Использование регистров:
        ;
        ;       R3= вычисляемое число
        ;       R4= старшие 32 бита (64 бита) числа
        ;       R5= счетчик выводимых цифр
        ;
ASC:
        PUHR    #^M<R3,R4,R5>   ;Сохранить регистры
        CLRQ    R4              ;Очистка R4 и R5
        MOVL    SIZE,R3         ;Получить вычисляемое число
NEXT1:  EDIV    #10,R3,R3,-(SP) ;Получить десятичную цифру (остаток)
        BEQL    NEXT2           ;Переход, если частное равно
        INCL    R5              ;Подсчет числа цифр
        BRB     NEXT1           ;Продолжить для следующей цифры
        ;
        ;Цифры загружены в стек в обратном порядке.
        ;Теперь они извлекаются, преобразуются в код СМ1700
        ;и затем выводятся.
        ;
NEXT2:  ADDL3   #^A/0/,(SP)+,R1 ;Цифра сформирована в коде СМ1700
        OUTCHAR R1              ;OUTCHAR выводит цифру
        SOBGEQ  R5,NEXT2        ;Продолжить до завершения
        POPR    #^M<R3,R4,R5>   ;Восстановить регистры
```

Рис. 3.3. Программа перекодировки и вывода целого числа

очищает два последовательных регистра: указанный в инструкции и следующий за ним по порядку.

Например, если SIZE содержит значение 265, то после трехкратного выполнения инструкции EDIV в цикле NEXT1 в стеке последовательно будут запомнены двоично-десятичные цифры 5, 6 и 2. В цикле NEXT2 символы извлекаются из стека, преобразуются в 8-битовый код и выводятся на видеотерминал в последовательности 2, 6 и 5.

3.3. ПОДПРОГРАММЫ И ПРОЦЕДУРЫ

Практика показывает, что большие монолитные программы трудно как разрабатывать, так и обслуживать. Программы, составленные из небольших самостоятельных частей, могут быть отлажены независимо одна от другой, что существенно повысит производительность труда программиста, надежность и удобство эксплуатации.

Подпрограммы позволяют создавать большие программы, которые легче обслуживать, при этом экономятся размеры программы (если отдельная функция должна выполняться не-

сколько раз в программе, то целесообразно реализовать эту функцию в виде подпрограммы). Однако совсем не обязательно, чтобы подпрограмма вызывалась более одного раза в программе. Последовательность инструкций должна реализовываться в виде подпрограммы всякий раз, когда это упрощает программирование или понимание программы.

Обращения к подпрограммам и передача аргументов определяются посредством стандартного механизма вызова-возврата. Набор инструкций СМ1700 содержит две формы вызова подпрограмм: вызов процедуры и обращение к подпрограмме. Решающее отличие между этими двумя формами заключается в механизме связи. Вызов процедуры является более мощным средством при обработке аргументов. При этом некоторые из общих регистров закреплены за аппаратными средствами вызова процедуры и для автоматической передачи управления и аргументов активно используется стек. Вызов подпрограммы, хотя и использует некоторые из регистров и стек, выполняется проще и быстрее.

Организация процедур. В СМ1700 общие средства вызова процедуры обеспечиваются инструкциями типа CALL (вызов процедуры). Архитектура вызова гарантирует следующие возможности:

1. Единый механизм передачи аргументов: вызванная процедура может определить число переданных аргументов и может адресоваться к ним с помощью смещений от фиксированного указателя, называемого «указателем аргумента» AP (Argument Pointer), в качестве которого служит регистр R13.

2. Общие регистры используемые процедурой, сохраняются автоматически аппаратными средствами.

3. Вызванная процедура может разместить свою локальную память в стеке и адресовать локальные переменные с помощью смещений от фиксированного указателя FP (Frame Pointer), называемого «указателем кадра», в качестве которого служит регистр R13.

4. Возврат выполняется единообразно: аппаратными средствами сохраненные регистры будут восстановлены и управление будет возвращено к инструкции, следующей за инструкцией CALL.

Таким образом, вызовы процедур — это в достаточной степени сложные реализации. Однако они помогают достичь лучшего структурирования программ. Еще более существенно то, что реализация вызова процедуры на архитектурном уровне устанавливает стандарт для всех компиляторов СМ1700. Следовательно, программа может вызывать процедуры, написанные на различных языках программирования.

Списки аргументов и инструкции вызова. В СМ1700 аргументы передаются вызванной процедуре в списке аргументов. Список аргументов — это массив двойных слов, в котором первое двойное слово в первом байте содержит счетчик числа следуемых за

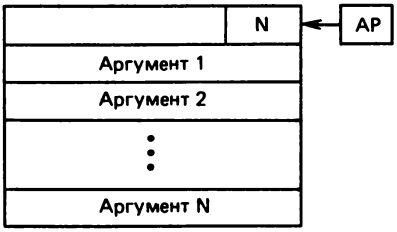


Рис. 3.4. Список аргументов

ним аргументов. Общий формат показан на рис. 3.4.

Каждый элемент в списке может быть значением или адресом. Передача данных процедуре посредством адреса называется «вызов по ссылке», в то время как передача непосредственно значений в списке аргументов — «вызов по значению». Список аргументов может быть размещен в статической памяти или загружен в стек перед вызовом. Поэтому существуют две формы инструкции вызова процедуры: CALLG и CALLS.

CALLG (вызов процедуры общего вида) воспринимает список аргументов в любом месте памяти. Следующий фрагмент показывает размещение списка аргументов ARLIST и формат инструкции CALLG, вызывающей процедуру MYPROG:

```
        <секция данных>
ARLIST: .LONG   2               ;Счетчик аргументов
        .LONG   600             ;Первый аргумент
        .LONG   84              ;Второй аргумент
        .
        .
START:  <секция программы>
        .
        .
        CALLG   ARLIST,MYPROG   ;Вызов процедуры MYPROG
```

Если повторное обращение к MYPROG необходимо с другими значениями аргументов, основная программа может использовать другой список аргументов или динамически переслать новые значения в исходный список.

CALLS (вызов процедуры стекового вида) воспринимает список аргументов, предварительно загруженный в стек. Следующий фрагмент показывает обращение к MYPROG с помощью инструкции CALLS:

```
PUSHL   #84             ;Загрузить в стек второй аргумент
PUSHL   #600            ;Загрузить в стек первый аргумент
CALLS   #2,MYPROG       ;Вызвать процедуру MYPROG
```

Следует отметить две важные детали относительно использования инструкции CALLS. Во-первых, аргументы загружаются в стек в обратном порядке, поскольку стек растет в сторону младших адресов. Во-вторых, инструкция CALLS сама загружает счетчик аргументов в стек перед передачей управления процедуре.

Перед передачей управления процедуре инструкция вызова любого типа загружает R12 (AP — указатель аргумента) адресом списка аргументов в памяти или стеке. Процедура получает доступ к аргументам через фиксированные смещения от AP. Первое слово, на которое указывает AP, всегда содержит число

следуемых за ним аргументов. Например, процедура MYPROG может получить предназначенные ей аргументы следующим образом:

```
MYPROG: .
        .
        MOVL    (AP),R1      ;Получить счетчик аргументов
        MOVL    4(AP),R2     ;Получить первый аргумент
        MOVL    8(AP),R3     ;Получить второй аргумент
        .
        .
        RET                  ;Возврат к вызвавшей программе
```

Существует ряд соглашений между использованием инструкций CALLG и CALLS. Первая имеет смысл там, где число аргументов и значение каждого аргумента фиксированы. Она выполняется обычно быстрее, потому что список аргументов размещается при загрузке программы и никаких действий не надо выполнять для загрузки аргументов в стек. В зависимости от числа вызовов, используемых в программе, та или иная форма вызова может быть эффективнее по объему занимаемой памяти. В СМ1700 инструкция загрузки значения в стек иногда короче, чем память, требуемая для самого значения, поскольку существует режим адресации короткого литерала. Например, инструкция загрузки в стек целого числа 9 требует два байта, в то время как двухсловный аргумент, содержащий число 9,— четыре байта. Еще более существенно то, что стековая форма вызова допустима для организации вложенных процедур, а также допускает рекурсию и повторную входимость.

Сохранение и восстановление регистров. Инструкции CALLG и CALLS передают управление процедуре по адресу, указанному во втором операнде. Для того чтобы сохранять регистры, первое слово вызванной процедуры всегда должно содержать маску сохранения, подобную той, которая используется в инструкциях PUSHR и POPR. Инструкция вызова проверяет маску и сохраняет в стеке регистры, соответствующие установленным разрядам маски. По соглашению в СМ1700 все регистры, используемые в процедуре, должны быть сохранены данной процедурой, за исключением R0 и R1, которые используются для возврата результатов в вызвавшую программу. Таким образом, при вызове процедур нет повода для беспокойства относительно сохранения или восстановления регистров, «проходящих» через вызываемые процедуры. Инструкция RET (возврат из процедуры) восстанавливает сохраненные регистры перед возвратом управления.

Каждая вызванная процедура имеет одинаковый базовый формат:

```
PROC:   .WORD   ^M<Rx,...,Ry>  ;Сохранить регистры
        .
        .
        <тело процедуры>
        .
        .
        RET                    ;Возврат к вызвавшей программе
```

```
        .SBTTL SORT  -- Процедура сортировки массива двойных слов
;++
;
;ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ:
;
;       Процедура упорядочивает массив целочисленных знаковых
;       двойных слов в порядк возрастания чисел
;
;ВЫЗЫВАЮЩАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ:
;
;       Вызов SORT при помощи CALLS или CALLG
;
;ВХОДНЫЕ ПАРАМЕТРЫ:
;
;       ARR(AP) - адрес массива длинных слов, подлежащих сортировке
;       ARRSIZE(AP) - адрес числа двойных слов в массиве
;
;ВЫХОДНЫЕ ПАРАМЕТРЫ И КОДЫ ЗАВЕРШЕНИЯ:
;
;       нет
;
;ПОБОЧНЫЕ ЭФФЕКТЫ:
;
;       все регистры, за исключением R1 и R0, сохраняются
;--
        ARR=4           ;Смещение к адресу массива
        ARRSIZE=8       ;Смещение к длине массива
;
;Следующая программа реализует "пузырьковую" сортировку, где
;меньшие значения "всплывают" (как пузырьки) вверх к вершине списка
;
SORT:
        .WORD   ^M<R3,R4>               ;Сохранить регистры
        SUBL3   #1,@ARRSIZE(AP),R1      ;Индекс последнего элемента
        MOVL    ARR(AP),R0              ;Базовый адрес массива
        CLRL    R3                      ;I - индекс первого элемента
10$:    ADDL3   #1,R3,R4                ;Инициализировать J к I+1
20$:    CMPL    (R0)[R3],(R0)[R4]       ;Первый элемент =< второму?
        BLEQ    30$                     ;Переход, если это так
;
;Найти последовательность двух чисел: сначала большее,
;а затем меньшее и поменять их местами
;
        PUSHL   (R0)[R3]                ;Сохранить ARR(I)
        MOVL    (R0)[R4],(R0)[R3]       ;Поменять ARR(I)
        POPL    (R0)[R$]                ;и ARR(J)
30$:
        AOBLSS  R1,R4,20$               ;Продвинуть J
        AOBLSS  R1,R3,10$               ;Продвинуть I
        RET                             ;Возврат
```

Рис. 3.5. Процедура сортировки

Воспользуемся примером сортировки (см. рис. 3.2) для иллюстрации правил написания процедур. Процедура SORT (рис. 3.5) будет вызываться со списком аргументов, содержащим два аргумента: адрес массива двойных слов, подлежащих сортировке, и адрес числа элементов в массиве. Этот пример также показывает стиль, которым рекомендуется пользоваться для документирования программ. В программах на ассемблере

02CM1700 принято, что результаты процедуры возвращаются в R0 и R1. Следовательно, эти регистры обычно не сохраняются и не восстанавливаются вызванной процедурой.

Одно из изменений, сделанных в этой программе (по сравнению с предыдущей программой сортировки на рис. 3.2),— введение локальных меток. Локальная метка — это символ вида N$, где N — целое число. Локальная метка определена только внутри блока локальных символов, в котором она находится. Границами блока являются любые две алфавитно-цифровые метки. Каждая новая алфавитно-цифровая метка начинает новый блок локальных символов, в котором локальные символы могут использовать те же самые номера. Другими словами, процедура, следующая за процедурой сортировки, также может использовать символы 10$, 20$ и 30$. Поскольку другая процедура может быть добавлена к процедуре сортировки только своей входной точкой, имеющей обычную метку, то локальные символы процедур оказываются вынужденно изолированными.

Кадр вызова. При исполнении инструкции RET в конце процедуры автоматически выполняются следующие действия:

1. Регистры, сохраненные при входе в процедуру, восстанавливаются.
2. Список аргументов удаляется из стека (если процедура была вызвана инструкцией CALLS).
3. Восстанавливается состояние процессора, и управление возвращается к инструкции, следующей за инструкцией вызова.

Другими словами, состояние основной программы восстанавливается, за исключением выходных результатов, которые могут быть записаны внутри процедуры или возвращены в регистры R0 или R1.

Инструкция RET может выполнить эти действия благодаря тому, что стек содержит всю необходимую информацию, называемую кадром вызова и создаваемую инструкцией CALL. Регистр R13 (FP — указатель кадра) при вызове процедуры загружается адресом кадра вызова. Следовательно, инструкция RET просто использует FP, чтобы определить место нахождения кадра вызова и восстановить предыдущее состояние.

Компонентами кадра вызова (рис. 3.6) являются:

1. Двойное слово адреса программы обработки исключительных ситуаций (CH). В нем вызывающая программа может запомнить адрес программы обработки ошибок, которая будет вызвана, если в процедуре возникнет исключительное состояние.
2. Сохраненное слово состояния процессора (PSW) вызывающей программы. Оно содержит биты разрешения, показывающие, как обрабатывать исключительные ситуации. (Коды условия NZVC запоминаются в виде нулей и не сохраняются после вызова.)
3. Копия маски сохранения регистров. Благодаря этому инструкция RET знает, какие регистры должны быть восстановлены.

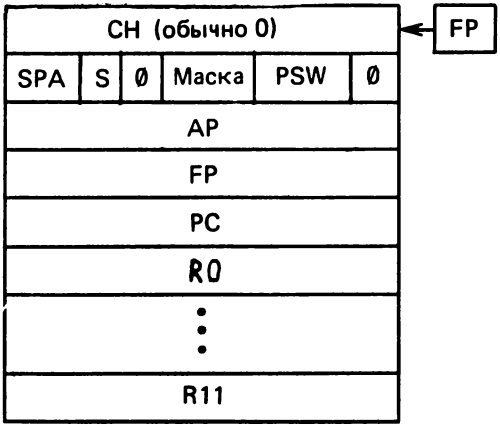


Рис. 3.6. Кадр вызова

4. Одиночный бит S, индицирующий, является ли этот вызов результатом CALLS (S=1) или CALLG (S=0). Если была использована инструкция CALLS, то инструкция RET удалит список аргументов из стека, опросив счетчик аргументов в первом двойном слове этого списка, чтобы определить число аргументов.

5. Двухразрядное поле SPA (выравниватель указателя стека). Инструкция вызова всегда выравнивает кадр вызова на границу двойного слова. Следовательно, если стек не был выровнен на момент вызова, то до трех пустых байтов может быть пропущено перед первым загружаемым в кадр вызова элементом. Поле SPA используется с целью запомнить, сколько байтов было пропущено, чтобы указатель стека мог быть восстановлен при уничтожении кадра вызова.

6. Все общие регистры, указанные в маске сохранения, располагаются в кадре вызова, начиная с регистра, имеющего самый большой номер (как в инструкции PUSHR). Всегда сохраняются PC, FP и AP.

Рассмотрим следующий пример. Предположим, что для вызова процедуры сортировки SORT (см. рис. 3.5) используется следующий фрагмент:

```
LSIZE:  .LONG   20          ;Размер массива
AGLIST: .BLKL   20          ;Массив из 20 элементов
          .
          .
MAIN:   <последовательность программного кода>
          .
          .
10$:    PUSHAL  LSIZE       ;Загрузить адрес длины массива
        PUSHAL  AGLIST      ;Загрузить адрес массива
        CALLS   #2,SORT     ;Вызвать процедуру сортировки
```

Пусть стек выровнен на границу двойного слова перед инструкцией PUSHAL (метка 10$). Состояние стека сразу после вызова показано на рис. 3.7. Вызванная процедура может использовать стек, и поэтому значение указателя стека SP может измениться. Указатель кадра FP всегда остается неизменным. Кадр вызова, таким образом, всегда может быть найден.

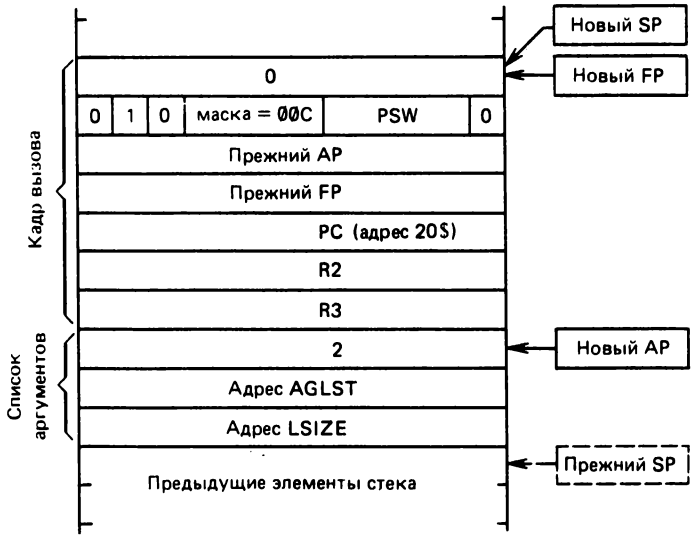


Рис. 3.7. Состояние стека после вызова процедуры SORT

Одним из достоинств фиксации FP является то, что процедура может разместить свои локальные переменные в стеке и адресовать их через фиксированные смещения от FP. Приведенная ниже процедура имеет две локальные переменные A и B, которые размещены в стеке. Пример показывает определение и адресацию этих переменных:

```
                A = -4                  ;Смещение до метки A
                B = -8                  ;Смещение до метки B
                  .
                  .
PROC:   .WORD   ^M<R3,R4,R9,R11>        ;Сохраняемые регистры
        SUBL    #8,SP                   ;В стеке выделить место для
                                        ;двух слоев двойной длины
        MOVL    R1,A(FP)                ;Запомнить R1 в A
        MOVL    R2,B(FP)                ;Запомнить R2 в B
                  .
                  .
        RET                             ;Возврат
```

Преимуществом такого способа фиксации является то, что как только память под локальные переменные отведена, стек по-прежнему может использоваться для промежуточного запоминания, не нарушая адресацию локальных переменных. Если бы начало стека процедуры не было зафиксировано в FP, то пришлось бы «вручную» отслеживать изменение смещения между указателем SP и памятью локальных переменных, что достаточно трудно, особенно когда одна процедура вызывает другую. Еще одно достоинство фиксации FP состоит в следующем: инструкция RET автоматически освобождает область стека, занятую локальными переменными.

Организация подпрограммы. Вызов процедуры используется как стандартный интерфейс между программами, написанными на любых языках. Следовательно, для программы, написанной на языке Фортран, можно, например, обратиться к процедуре, написанной на языке Паскаль или на языке ассемблера.

Однако при программировании на ассемблере иногда желательно вызвать короткую процедуру (известную как «подпрограмма» в архитектуре СМ1700) без накладных расходов универсального вызова. При этом нет автоматического сохранения или восстановления регистров через стек; вызванная подпрограмма сама должна заботиться об этом.

Инструкции BSBB и BSBW (переход к подпрограмме) используются для вызова коротких подпрограмм. Точно так же, как инструкции переходов BRB и BRW, эти инструкции выполняют переходы к подпрограммам внутри байтных или словных смещений относительно текущей ячейки. Когда инструкция BSBB или BSBW выполняется, она просто загружает в стек содержимое регистра PC (адрес инструкции, к которой следует вернуться) и переходит к подпрограмме. Для возврата подпрограмма выполняет инструкцию RSB (возврат из подпрограммы), которая всего лишь извлекает адрес возврата из стека и возвращает управление. Поскольку инструкция BSB не использует указатель кадра, подпрограмма должна обеспечить сохранение и восстановление регистров и приведение стека к его первоначальному состоянию перед выполнением инструкции RSB. При организации подпрограмм аргументы и результаты передаются обычно через общие регистры.

В процедуре COMPUT (рис. 3.8) вычисляется степенная сумма элементов массива, заданного адресом ARR и числом элементов N:

$$ARR[0]^0 + ARR[1]^1 + \ldots + ARR[N]^N$$

Здесь процедура COMPUT инструкцией BSBB обращается к подпрограмме POWER (рис. 3.9) для возведения каждого элемента в соответствующую степень.

Помимо инструкций типа BSB существует инструкция JSB (универсальный переход к подпрограмме), которая подобно инструкции JMP действует в пределах всего адресного пространства и допускает различные режимы адресации при вычислении адреса подпрограммы. Подпрограммы, вызванные инструкцией JSB, возвращают управление также с помощью инструкции RSB.

Рекурсия и повторная входимость. Часто для подпрограммы или процедуры полезной является рекурсия, т. е. способность вызывать саму себя. Такие программы упрощают программирование многих математических и синтаксических алгоритмов.

```
        .SBTTL  COMPUT - Подсчет степенной суммы массива
;++
;
;ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ:
;
;       Данная программа вычисляет значение
;            0        1        2              N
;       ARR[0]  + ARR[1]  + ARR[2]  + ...  + ARR[N]
;
;ВЫЗЫВАЮЩАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ:     CALLS или CALLG
;
;ВХОДНЫЕ ПАРАМЕТРЫ:
;
;       ARR(AP)  - Адрес маассива
;       SIZE(AP) - Число элементов в массиве
;       RSLT(AP) - Адрес двойного слова для
;                  вычисленного результата
;
;ВЫХОДНЫЕ ПАРАМЕТРЫ:
;
;       RSLT(AP)  -  Адрес результата
;
;ПОБОЧНЫЕ ЭФФЕКТЫ:   не сохранен регистр R1
;
;--
;
        ARR=4           ;Смещение к адресу массива
        SIZE=8          ;Смещение к размеру массива
        RSLT=12         ;Смещение к адресу двойного
                        ;слова результата
;
;ИСПОЛЬЗОВАНИЕ РЕГИСТРОВ:
;
;       R2 = значение текущего элемента массива
;       R3 = адрес следующего элемента массива
;       R4 = значение степени
;       R1 = значение текущего элемента суммы
;       R5 = аккумулятор суммы
;
COMPUT:
        .WORD   ^M<R3,R4,R5,R1,R2>
        MOVL    ARR(AP),R3      ;Адрес первого элемента
        CLRQ    R4              ;Очистка R4 и R5 (степень и сумма)
10$:    MOVL    (R3)+,R2        ;Текущий элемент массива
        BSBB    POWER           ;Вычислить текущий элемент суммы
                                ;(возведение в степень)
        ADDL    R1,R5           ;Прибавить этот элемент к сумме
        AOBLSS  SIZE(AP),R4,10$ ;Продолжить со следующими
                                ;элементом и степенью
        MOVL    R5,@RSLT(AP)    ;Запомнить результат
        RET
```

Рис. 3.8. Процедура подсчета суммы массива

Для того чтобы программа была рекурсивной, она не должна иметь статической памяти. Все локальные переменные должны размещаться в стеке так, чтобы переменные от одного вызова не модифицировались при другом рекурсивном вызове.

Классическим примером рекурсии является вычисление N! (факториала числа N), который, как известно, определяется формулой $N! = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1$.

```
;
;ПОДПРОГРАММА ВОЗВЕДЕНИЯ В ЗАДАННУЮ СТЕПЕНЬ
;
;Входные параметры:
;
;       R2 = значение элемента массива
;       R4 = возведение в степень
;
;Выходные параметры:
;
;       R1 = R2 в степени R4
;
POWER:
        MOVL    #1,R1           ;Примем R1=1 (нулевая степень)
        PUSHL   R4              ;Сохранить R4 (заданная степень)
        BLEQ    30$             ;Переход,если степень < или = 0
        MOVL    R2,R1           ;Значение для первой степени
        BRB     20$             ;Начать цикл с конца
10$:    MULL    R2,R1           ;Вычислить следующую степень
20$:    SOBGTR  R4,10$          ;Продолжить цикл до завершения
30$:    POPL    R4              ;Восстановить сохраненный регистр
        RSB                     ;Возврат к основной программе
```

Рис. 3.9. Подпрограмма возведения в степень

Значение факториала нуля определяется равным единице и поэтому рекурсивное определение N! можно записать в следующем виде:

$N! = N \cdot (N-1)!$ для $N > 0$,

$N! = 1$ для $N = 0$.

На языке Паскаль рекурсивную программу вычисления N! можно записать следующим образом:

```
FUNCTION FACT(N: INTEGER): INTEGER;
BEGIN   IF N<=1 THEN FAST:=1
        ELSE  FAST:=N*FAST(N-1)
END;
```

Имеется два способа представить эту программу на ассемблере в зависимости от использования инструкций связи CALLS (рис. 3.10) или BSB (рис. 3.11).

```
BEGIN:  .WORD   0               ;Входная маска главной процедуры
        PUSHL   N               ;Число для вычисления факториала
        CALLS   #1,FACT         ;Вызов процедуры вычисления факториала
1$:     RET                     ;Возврат с результатом в R1
;
;РЕКУРСИВНАЯ ПРОЦЕДУРА ВЫЧИСЛЕНИЯ N!
;
FACT:    .WORD   ^M<>           ;Нет сохраняемых регистров
        MOVL    #1,R1           ;Приняв N=0, получить N!=1
        MOVL    4(AP),R0        ;Копировать N
        BEQL    10$             ;Выход, если N=0
;
;Создать кадр вызова с N-1 для следующего обращения,
;чтобы вычислить (N-1)!
;
        SUBL3   #1,R0,-(SP)     ;Загрузить N-1 в стек
        CALLS   #1,FACT         ;Рекурсивный вызов
5$:     MULL    R0,R1           ;Вычислить N*(N-1)!
10$:    RET  R0                 ;Возврат (к 5$ или к 1$)
```

Рис. 3.10. Рекурсивная процедура вычисления факториала

```
BEGIN:  MOVL    N,R1            ;Число для вычисления факториала
        BSBB    FACT            ;Вычислить N!
1$:     RBS                     ;Возврат к вызвавшей программе
;
;Рекурсивная подпрограмма для вычисления N!. Число N передается
;подпрограмме в R1. На каждом уровне рекурсии N сохраняется в
;R2 и N-1 передается рекурсивно вызываемой подпрограмме
;для вычисления N-1!
;
FACT:   PUSHL   R2              ;Сохранить R2
        MOVL    R1,R2           ;Копировать N
        BNEQ    10$             ;Продолжить, если N не равно 0
        MOVL    #1,R1           ;При N=0 возврат с результатом N!=1
        BRB     20$             ;Выход
10$:    DECL    R1              ;Вычислить N-1
        BSBB    FACT            ;Рекурсивный вызов
15$:    MULL    R2,R1           ;Вычислить N*(N-1)!
20$:    POPL    R2              ;Восстановить R2
        RSB                     ;Возврат (к 15$ или к 1$)
```

Рис. 3.11. Рекурсивная подпрограмма вычисления факториала

В обоих случаях используется стек для «раскручивания» рекурсии и регистр R1 для возврата значения N!. В первом CALLS получает N через кадр вызова, в то время как во втором BSB получает N через R1.

Хотя рекурсивное определение N! является удобным в математическом смысле, для ЭВМ это не лучший способ. Нерекурсивное решение (рис. 3.12) проще и быстрее. При этом обеспечивается более экономное использование области стека.

```
FACT:   MOVL    #1,R1           ;Инициализировать факториал
        TSTL    N               ;Опрос N
        BEQL    20$             ;При N=0 возврат с N!=1
        MOVL    #1,R0           ;Инициализировать счетчик
10$:    MULL    R0,R1           ;Вычислить N*(N-1)!
        AOBLEQ  N,R0,10$        ;Продолжать до завершения
20$:    RSB                     ;Возврат к вызывавшей программе
```

Рис. 3.12. Подпрограмма вычисления факториала

Главным недостатком рекурсивных программ является то, что они требуют значительных размеров области стека для вложенных кадров вызова и переменных. Поэтому следует разумно подходить к использованию рекурсии. Возможно, более целесообразным примером рекурсивной программы будет подпрограмма преобразования целого числа в 8-битовый код (рис. 3.13), которая является разновидностью программы, обсужденной ранее (см. рис. 3.3).

Таким образом, рекурсия допускает существование нескольких одновременных вызовов процедуры, каждый из которых находится в своей собственной стадии выполнения. Это возможно, поскольку каждый вызов имеет собственную локальную информацию состояния, содержащуюся в кадре вызова и локальной памяти. Всякий раз, когда выполняется рекурсивный вызов, возникает новое состояние из-за создания в стеке кадра вызова и тем самым сохраняется состояние вызывающей программы.

В мультипрограммных системах это свойство особенно полезно для программ общего использования в режиме разделения

```
;Вывод целого числа в 8-битовом коде СМ1700
;
;Использование регистров:
;
;       R3 = вычисляемое число
;       R4 = старшие 32 бита числа
;       R5 = количество выводимых цифр
;
ASC:    PUSHR   #^M<R3,R4>          ;Сохранить регистры
        CLRL    R4                  ;Обнулить старшие биты для EDIV
        MOVL    NUMBER,R3           ;Получить выводимое число
        BSBB    NUM                 ;Вызвать подпрограмму вывода числа
        POPR    #^M<R3,R4>          ;Восстановить регистры
        RSB                         ;Возврат
;
;Рекурсивная подпрограмма вывода числа в коде СМ1700. Каждый уровень
;выделяет самую младшую цифру и выполняет рекурсивный вызов, если
;результат не 0. Когда последняя цифра получена,выполняется возврат,
;раскручиваясь назад через уровни и выписывая цифры в порядке от
;старшей к младшей.
;
NUM:
        EDIV    #10,R3,R3,-(SP)     ;Загрузить в стек младшую цифру
        BEQL    10$                 ;Переход, если выполнено
        BSBB    NUM                 ;Рекурсия для следующей цифры
10$:    ADDL3   #^A/0/,(SP)+,R1     ;Поместить в R1 цифру в коде СМ1700
        OUTSTHAR R1                 ;Макрокоманда вывода цифры
        RBS                         ;Возврат к вызывавшей программе
```

Рис. 3.13. Подпрограмма вывода целого числа в 8-битовом коде

времени, т. е. для нескольких пользовательских процессов желательно иметь возможность выполнять процедуру, не создавая ее копии для каждого процесса. Такое коллективное пользование помогает экономить объем занимаемой памяти, позволяя большему числу пользователей разместить программы в памяти и таким образом улучшить характеристики системы. Наибольший выигрыш получается от разделения системных программ общего пользования: редакторов, компиляторов, компоновщиков и др. Например, для многих пользователей, выполняющих редактирование, в памяти необходима только одна копия программы-редактора.

Процедуры, которые одновременно могут быть разделены многими пользователями, называются повторно-входимыми. Несколько различных пользовательских процессов могут находиться в своей стадии выполнения в различных местах внутри повторно-входимой процедуры. Для того чтобы процедура была повторно-входимой, ее образ в оперативной памяти (секция программы и секция данных) не должен изменяться в процессе выполнения. Любые переменные, которые модифицируются, должны располагаться в стеке или общих регистрах, прежнее состояние которых сохранено в стеке. Поскольку каждый пользовательский процесс имеет свой собственный счетчик инструкций, набор общих регистров, указатель стека и стек, то несколько пользовательских программ могут находиться в процессе выполнения

одной и той же повторно-входимой программы. Если одна программа пользователя прерывается, а другая начинает выполнение той же программной последовательности, то потери информации не возникает, поскольку модифицируемые данные изолированы в области стека каждого пользователя.

Кроме упомянутых системных программ сама операционная система МОС ВП, которой посвящены гл. 5 и 6, также компонуется из повторно-входимых программ. Все процессы полностью разделяют операционную систему, и многие из них могут выполнять ее процедуры в одно и то же время.

3.4. МАКРОКОМАНДЫ

В программах на языке ассемблера часто возникают некоторые повторяющиеся последовательности инструкций. Возможности макрокоманд языка ассемблера позволяют описать повторяющуюся последовательность инструкций и присвоить ей конкретное имя макрокоманды, так называемый «макрос». Макрос может рассматриваться как тело «открытой» подпрограммы в том смысле, что исходные инструкции воспроизводятся в каждом месте программы, где этот макрос используется.

Например, если часто требуется очистка регистров R0 – R11, то, создав определение макроса CLRREG:

```
CLRO    R0          ;Очистить R0 - R3
CLRO    R4          ;Очистить R4 - R7
CLRO    R8          ;Очистить R8 - R11
```

в программе можно воспользоваться именем CLRREG, как если бы это было машинной инструкцией. При каждом обнаружении имени CLRREG в тексте программы ассемблер генерирует три инструкции, содержащиеся в этом макросе.

Замена одной символьной строки (имя макроса) на несколько символьных строк (тело макроса) называется макрорасширением. Ассемблер, имеющий такие возможности, называется макроассемблером.

Макрос должен быть определен до его первого использования. Директива .MACRO начинает определение макроса (макроопределение), в то время как директива .ENDM информирует макроассемблер о конце исходного текста, который будет включен в исходную программу (макрорасширение), в том месте, где встречается имя макроса (макровызов). В листинге распечатываются, как правило, только макровызовы; для распечатки макрорасширений необходима директива управления листингом .SHOW с аргументом ME.

Всякий раз, когда макрос CLRREG используется, он генерирует одни и те же инструкции. Иногда полезно иметь такие макросы, которые позволяют создавать разные последовательности инструкций в зависимости от каких-то параметров. Это может быть сделано за счет использования формальных аргументов внутри макроопределения. В этом случае при создании конкретного макрорасширения формальные аргументы замещаются на фактические аргументы, определенные в данном макровызове.

Например, использовав формальный аргумент INDEX, можно создать обобщенный макрос загрузки квадрослова в стек:

```
.MACRO  RUSHQ   INDEX=R0
MOVQ    INDEX,-(SP)
.ENDM   RUSHQ
```

Всякий раз, когда макрос вызывается, указанный фактический аргумент заменяет формальный аргумент INDEX в макрорасширении. Так, программа, содержащая макровызов PUSHQ TEMP[R8], генерирует инструкцию MOVQ TEMP[R8], −(SP).

Когда требуется несколько аргументов, они разделяются запятыми. В этом случае фактические аргументы должны указываться в том же самом порядке, что и формальные.

Ключевые параметры. Ключевой параметр — это имя формального аргумента вместе с его значением по умолчанию. Общий формат макроопределения:

```
.MACRO  ИМЯ_МАКРОСА     Арг1,Арг2,...,АргN
    .
    .   тело макроса
    .
.ENDM   ИМЯ_МАКРОСА
```

Здесь каждый аргумент имеет вид «ключ-умолч», где правая часть определяет значение данного аргумента по умолчанию.

Например, если макрос PUSHQ часто используется для загрузки R0 и R1, то он может быть определен с таким умолчанием:

```
.MACRO  RUSHQ   INDEX
MOVQ    INDEX,-(SP)
.ENDM   RUSHQ
```

Теперь использование макроса PUSHQ в макровызове без аргументов вырабатывает инструкцию MOVQ R0,−(SP). Для использования макроса PUSHQ с другим аргументом в макровызове необходимо использовать конструкцию «ключ-факт», где правая часть определяет фактический аргумент. Например, макрос PUSHQ INDEX=TEMP[R8] создаст инструкцию MOVQ TEMP[R8],−(SP).

Ключевые параметры наиболее полезны в макросах со многими аргументами, потому что в макровызовах аргументы могут быть указаны в любом порядке.

Создание локальных меток. Часто возникает необходимость иметь внутри макросов метки. Если программист определит метки в макроопределении, это приведет к тому, что будут появляться многократно определенные метки, поскольку каждый макровызов создает макрорасширение с той же самой меткой. Чтобы избежать такой ошибки, макроассемблер предоставляет средства для создания уникальных (неповторяющихся) локальных меток внутри макрорасширений.

Для иллюстрации этой возможности рассмотрим макрос, который реализует псевдоинструкцию JMPZ (переход, если ноль):

```
    .MACRO  JMPZ    DATA,GOAL,?X
    TSTL    DATA                    ;DATA=0?
    BNEQ    X                       ;Продолжить, если нет
    JMP     GOAL                    ;Иначе перейти к GOAL
X:
    .ENDM   JMPZ
```

Заметим, что макрос может содержать комментарии. Помещение знака вопроса перед формальным аргументом определяет локальную метку. При расширении макроса ассемблер создает новую локальную метку, если соответствующее место фактического аргумента пусто. Если же соответствующий фактический аргумент определен, то ассемблер заменяет формальный аргумент на фактический.

Локальные метки, создаваемые макроассемблером, начинаются с 30000$ и заканчиваются 65535$. Каждый раз, когда макроассемблер генерирует новую локальную метку, номерная часть метки увеличивается на единицу. Следовательно, каждый раз генерируется уникальная метка, которая не пересекается с локальными метками, явно определенными пользователем.

Следующий пример показывает двукратное использование макроса JMPZ и генерируемый текст:

```
                                    TSTL    S
                                    BNEQ    30000$
                                    JMP     HERE
JMPZ    S,HERE ------>  30000$:
JMPZ    T,THERE                     TSTL    T
                                    BNEQ    30000$
                                    JMP     THERE
                        30001$:
```

Заметим, что автоматически создаваемые локальные метки имеют зону действия между двумя обычными метками, создаваемыми пользователем.

Макровызовы внутри макроопределений. Для создания сложных макросов часто удобно использовать макрос внутри определения другого макроса. Другими словами, если макрос был ранее определен, то он может быть вызван другим макросом так же, как если бы он был частью системы команд.

Рассмотрим простые макросы загрузки в стек и извлечения из стека учетверенного слова:

```
MOVQ    B,-(SP)                 MOVQ    (SP)+,B
```

Эти макросы теперь можно использовать для написания более сложного макроса, который позволяет менять местами содержимое двух учетверенных слов:

```
.MACRO  SWAPQ   B,C
PUSHQ   B
MOVQ    C,B
POPQ    C
.ENDM   SWAPQ
```

Как известно, инструкции PUSHL и POPL уже существуют в ассемблере CM1700 (POPL в действительности является ассемблерно-определенным макросом). Добавив к ним PUSHQ и POPQ, достаточно просто теперь модифицировать макрос перестановки SWAP так, чтобы один из аргументов был типом данных L или Q. При этом имя операции создается посредством объединения строки PUSH со значением аргумента, содержащего индикатор типа L или Q. Такая операция конструирования единой строки из двух строк называется конкатенацией.

Конкатенация аргументов. Оператор конкатенации аргументов (апостроф) объединяет аргумент макроса с неким постоянным текстом. Апостроф может предшествовать имени формального аргумента или следовать за ним в исходном макросе. Если имя формального аргумента в макроопределении предшествует

апострофу (или следует за ним), то в макрорасширении текст перед апострофом (или после него) объединяется с фактическим аргументом; сам же апостроф в макрорасширении не появляется.

Следующий пример иллюстрирует макрос перестановки SWAP, обобщенный посредством включения формального аргумента TYPE:

```
.MACRO  SWAP    B,C,TYPE
PUSH'TYPE       B
MOV'TYPE        C,B
POP'TYPE        C
 .ENDM  SWAP
```

Этот макрос меняет местами два слова учетверенной или удвоенной длины, используя стек как промежуточную память. Ниже показаны два макровызова перестановки и генерируемые ими макрорасширения:

```
                        PUSHL   A
                        MOVL    B,A
SWAP    A,B,L ------->  POPL    B
SWAP    C,D,Q           PUSHQ   C
                        MOVQ    D,C
                        POPQ    D
```

Директивы повторения. При написании программ часто возникает необходимость создания таблицы значений. Директива «блок повторения», определяемая обозначениями .REPT и .ENDR, обеспечивает необходимое действие. Например, чтобы создать таблицу из десяти элементов, каждый из которых является байтом и содержит число 5, достаточно написать:

```
.REPT   10
.BYTE   5
.ENDR
```

Более общей является директива «неопределенный блок повторения». Она предусматривает задание списка аргументов. Общий формат директивы:

```
.IRP    СИМВОЛ,<список аргументов>
  .
  . <блок повторения>
  .
.ENDR
```

Макроассемблер копирует блок повторения столько раз, сколько имеется аргументов, заменяя параметр «символ» на очередной фактический аргумент. Список фактических аргументов должен быть заключен в угловые скобки, а в качестве разделителей должны использоваться запятые. Например, следующий неопределенный блок повторения загружает в стек ряд двойных слов:

```
.IRP    INDEX,<VALUE,R1,FRG,SQROOT>
PUSHL   INDEX
.ENDR
```

Макроассемблер раскрывает этот блок следующим образом:

```
PUSHL   VALUE
PUSHL   R1
PUSHL   FRG
PUSHL   SQROOT
```

Более сложный пример, в котором список передается макросу как аргумент, имеет место при генерации оператора выбора. Макрос CASE автоматически генерирует таблицу выбора из списка приемников перехода и вычисляет предел индекса:

```
        .MACRO CASE     SRC,DISP,TYPE=B,BASE=#0,?TBL,?ENDTBL
        CASE'TYPE       SRC,BASE,#<<ENDTBL-TBL>/2-1
TBL:    .IRP    DST,<DISP>
        .WORD   DST-TBL
        .ENDR
ENDTBL:
        .ENDM   CASE
```

В макросе используется несколько приемов, уже описанных ранее, а именно:

конкатенация для формирования одной из псевдоинструкций CASEB, CASEW или CASEL;

умолчание, где инструкцией по умолчанию является CASEB, а базой по умолчанию — ноль;

неопределенный повтор для формирования таблицы смещений, следуемой за инструкцией выбора;

автоматически создаваемые локальные метки.

Например, оператор

```
CASE    EXAMP,<A,B,C>TYPE=W
```

вызывает генерацию программы

```
        CASE    EXAMP,#0,#<<30001$-30000$>/2-1
30000$:
        .WORD   A-30000$
        .WORD   B-30000$
        .WORD   C-30000$
30001$:
```

Условное ассемблирование. Директивы условного ассемблирования позволяют в зависимости от каких-либо условий или транслировать какой-то участок программы, или пропускать его (не транслировать).

Формат директивы условного блока ассемблирования.

```
.IF     УСЛОВИЕ АРГУМЕНТ(Ы)
 .
 . <условные операторы>
 .
.ENDC
```

Если аргумент удовлетворяет указанному условию, то ассемблер генерирует операторы, содержащиеся внутри блока. Если условие не удовлетворяется, операторы не генерируются. Перечень условий и аргументов приведен в табл. 3.1.

В качестве примера использования условного ассемблирования рассмотрим макрос CALL, который реализует обобщенный вызов процедуры. Аргументы макроса состоят из имени вызываемой процедуры ROUT и не более десяти двухсловных параметров P1, ..., P10 для процедуры. Эти параметры загружаются в стек в обратном порядке. Поскольку макрос может иметь различное число аргументов, то используется условие NB, чтобы исключить загрузку в стек параметров, которые не указаны:

```
COUNT=0                 ;Аргументы в стек еще не загружены
PUSHL   ARG             ;Загрузить аргумент в стек
COUNT=COUNT+1           ;Счет загруженных аргументов
CALLS   #COUNT,ROUT     ;Вызов ROUT
```

Переменная COUNT используется для подсчета числа аргументов, загружаемых в стек для инструкции CALLS. С помощью директивы .IRP аргументы опрашиваются в обратном порядке и загружаются в стек, если они имеются. Например, оператор

```
CALL    DIZZ,R1,ARRAY2[R4],PEAN#8,@IND
```

генерирует следующую программную последовательность:

```
PUSHL   @IND            ;Загрузить в стек пятый аргумент
PUSHL   #8              ;Загрузить в стек четвертый аргумент
PUSHL   PEAN            ;Загрузить в стек третий аргумент
PUSHL   ARRAY2[R4]      ;Загрузить в стек второй аргумент
PUSHL   R1              ;Загрузить в стек первый аргумент
CALLS   #5,DIZZ         ;Вызов процедуры DIZZ
```

В момент ассемблирования инструкции CALLS переменная COUNT будет иметь значение 5.

Внутри условного блока можно использовать поддирективу .IFF, по которой часть блока ниже нее транслируется, если условие в директиве .IF (при входе в блок) не удовлетворяется. Таким образом, поддиректива .IFF позволяет разделить блок условной трансляции на два подблока, один из которых транслируется.

Внутри одного условного блока (внешнего) можно использовать другой (внутренний). При этом условие транслируемости внутреннего блока проверяется только в том случае, если транслируется часть внешнего блока, непосредственно предшествующая внутреннему блоку. Глубина вложенности может быть любой.

Таблица 3.1. **Условия и аргументы директивы .IF**

Мнемоника условия		Условие трансляции блока		Тип аргумента
прямого	обратного	прямое	обратное	
EQ	NE	Аргумент = 0	Аргумент ≠ 0	Арифметико-логическое выражение
GT	LE	Аргумент > 0	Аргумент ≤ 0	
LT	GE	Аргумент < 0	Аргумент ≥ 0	
DF	NDF	Аргумент определен	Аргумент не определен	Логическое выражение
B	NB	Аргумент отсутствует	Аргумент присутствует	Аргумент макровызова
IDN	DIF	Аргументы идентичны	Аргументы различны	Два аргумента макровызова

Глава 4. ИНСТРУКЦИИ ОБРАБОТКИ ПРОСТЫХ И СЛОЖНЫХ СТРУКТУР ДАННЫХ

4.1. АРИФМЕТИКА ЦЕЛЫХ ЧИСЕЛ РАЗНОЙ ДЛИНЫ

Часто необходимо преобразовать какой-то элемент данных из одного типа в другой для того, чтобы можно было использовать его в соответствующей арифметической операции. Инструкции преобразования (табл. 4.1) позволяют выполнить пересылку с преобразованием знаковых данных.

Более короткий тип знаковых данных преобразуется в более длинный, выполняется расширение знака, т. е. самый старший разряд, который является знаковым, дублируется в старших разрядах более длинного операнда-приемника, а младшие разряды копируются. Рассмотрим следующее преобразование байта в слово:

```
A:    .BYTE    ^B11010011    ;Исходный байт
B:    .WORD    0             ;Конечный результат
      CVTBW    A,B           ;Преобразование
```

После выполнения инструкции преобразования содержимое операндов A и B будет следующим: A = D3, B = FFD3. Младшие семь разрядов байта A (его значение) копируются без изменения в семь младших разрядов слова B, а знаковый разряд байта A распространяется влево, заполняя старший байт слова B.

При преобразовании от более длинного к более короткому типу данных знаковый разряд также копируется, но большее число усекается. Усечение иллюстрируется в следующем преобразовании:

```
A:    .BYTE    0                    ;Байт результата
B:    .WORD    ^B1111111101001111   ;Исходное слово
      CVTWB    B,A                  ;Преобразование
```

Таблица 4.1. **Инструкции преобразования целых чисел**

Инструкция	Мнемоника	Операнды
Преобразование байта в слово	CVTBW	Байт, слово
Преобразование байта в двойное слово	CVTBL	Байт, двойное слово
Преобразование слова в байт	CVTWB	Слово, байт
Преобразование слова в двойное слово	CVTWL	Слово, двойное слово
Преобразование двойного слова в байт	CVTLB	Двойное слово, байт
Преобразование двойного слова в слово	CVTLW	Двойное слово, слово

После инструкции преобразования содержимое операндов B и A будет следующим: B=FF4F, A=CF. В данном случае только знаковый разряд операнда B сохраняется в операнде A. Если бы старший байт состоял не из всех единиц, как показано в примере, то произошло бы переполнение целого числа. При использовании инструкций преобразования необходимо быть особенно внимательным, когда поле приемника меньше, чем поле источника. Переполнение целого числа будет возникать всякий раз, когда не все старшие разряды источника (отсекаемые разряды) совпадают с результирующим знаковым разрядом поля приемника.

Обычным при использовании таких инструкций, особенно преобразующих меньшее в большее, является перемещение байта или слова данных в регистр для выполнения арифметических операций. Если бы данные были пересланы в регистр инструкцией MOV (пересылка), то старшие разряды регистра сохранили бы свои прежние значения.

Для расширения беззнаковых типов данных существуют три инструкции «Пересылка с лидирующими нулями», которые расширяют нулями байты в слова (MOVZBW), байты в двойные слова (MOVZBL) и слова в двойные слова (MOVZWL). Эти инструкции также обычно используются, чтобы переслать короткие данные в регистр, где старшие разряды заменяются нулями. Например, вместо программы

```
CLRL    R0              ;Очистить регистр R0
MOVB    A,R0            ;Переслать байт A в R0<7:0>
```

можно написать

```
MOVZBL  A,R0            ;Переслать расширенный нулями
                        ;байт A в регистр R0
```

Контекст операнда определяется типом данных этого операнда. Таким образом, в инструкции MOVZBL первый операнд имеет контекст байта, а второй операнд — контекст двойного слова. После выполнения инструкции MOVZBL (R0)+,(R2)+ значение R0 увеличивается на 1, в то время как R2 увеличивается на 4.

Для хранения сверхбольших операндов в СМ1700 используется, как правило, несколько смежных двойных слов. Для 32-разрядной арифметики диапазон знаковых чисел простирается от −2147483648 до +2147483647; для 64-разрядной арифметики диапазон расширяется от -2^{63} до $2^{63}-1$. Однако какой бы ни был предоставлен диапазон, всегда найдутся задачи, которым потребуется еще больший диапазон.

Сложение и вычитание в увеличенном диапазоне выполняются инструкциями ADWC (сложение с переносом) и SBWC (вычитание с переносом), использующими бит C в регистре PSW. Чтобы сложить два 64-разрядных числа, необходимы две инструкции:

сложение младших разрядов и сложение старших 32 разрядов с переносом от первого сложения.

Например, если имеются два операнда B и C каждый размером в учетверенное слово, то их сложение можно выполнить следующим образом:

```
B:   .LONG   4321      ;Менее значащие разряды операнда B
     .LONG   6765      ;Более значащие разряды операнда B
C:   .LONG   1212      ;Менее значащие разряды операнда C
     .LONG   7878      ;Более значащие разряды операнда C
     .
     .
     ADDL    B,C       ;Сложить менее значащие разряды B и C
     ADWC    B+4,C+4   ;Сложить более значащие разряды B и C
                       ;с битом переноса от младших частей
```

Для чисел еще большего диапазона могут быть добавлены дополнительные инструкции ADWC.

В СМ1700 операции умножения и деления с фиксированной запятой реализованы следующим образом. Имеются не только традиционные инструкции MUL (умножение) и DIV (деление) в двух- и трехоперандных форматах, но также расширенные формы умножения EMUL и деления EDIV. Инструкция EMUL (расширенное умножение) перемножает два 32-разрядных значения, вырабатывая 64-разрядное произведение, и еще прибавляет другое 32-разрядное значение к результату. Инструкция EMUL имеет следующий формат: EMUL множитель, множимое, слагаемое, произведение.

В языках программирования высокого уровня эта инструкция эффективно реализует операторы вида A = B + C*D.

Инструкция EDIV (расширенное деление) была рассмотрена ранее в 3.2. Она часто используется, когда требуется остаток от деления, потому что обычные инструкции целочисленного деления вырабатывают только частное.

Наконец, существуют инструкции сдвигов для двойных и учетверенных слов. Эти инструкции позволяют выполнять арифметический сдвиг (влево или вправо) слов удвоенной или учетверенной длины с помощью инструкции ASH (арифметический сдвиг), а также циклический сдвиг слов удвоенной длины с помощью инструкции ROTL (ротация двойного слова).

4.2. БИТЫ И БИТОВЫЕ ПОЛЯ

Инструкции обработки битов могут быть разделены на три группы. Первая включает инструкции, которые выполняют логические функции над битами, содержащимися внутри байта, слова или двойного слова. Вторая группа содержит инструкции, которые тестируют (опрашивают) одиночный бит внутри двойного слова, чтобы определить, будет ли выполнен переход. Третья группа состоит из инструкций, которые имеют дело с произвольными битовыми строками.

Таблица 4.2. **Инструкции логической обработки битов**

Наименование	Мнемоника	Алгоритм
Установ битов	BIS{B,W,L}2 M, D	D←D∨M
	BIS{B,W,L}3 M, S, D	D←S∨M
Сброс битов	BIC{B,W,L}2 M, D	D←D&$\bar{M}$
	BIC{B,W,L}3 M, S, D	D←S&$\bar{M}$
Тест битов	BIT{B,W,L} M, S	S&M
Исключающее ИЛИ	XOR{B,W,L}2 M, D	D←D∀M
	XOR{B,W,L}3 M, S, D	D←S∀M

Логическая обработка битов. Инструкции BIS (установ битов), BIC (сброс битов) и XOR (исключающее ИЛИ) являются общим набором логических операций, которые манипулируют байтами, словами, двойными словами и имеют две модификации: два операнда и три операнда (табл. 4.2).

При использовании этих инструкций обычно рассматривают первый операнд M как маску, разряды которой воздействуют на второй операнд (S или D) согласно коду инструкции. Результат записывается на место операнда-приемника D. Например, инструкция BICB #5,R4 сбрасывает биты 0 и 2 в регистре R4, поскольку эти биты установлены в операнде-маске и их двоичный эквивалент равен десятичному числу 5.

Инструкция BISB #3,R4 устанавливает биты 0 и 1 в регистре R4, поскольку маска имеет двоичный эквивалент 00...0011.

Инструкция BIT (тест битов) проверяет, установлен ли в единицу в операнде-приемнике любой из тех битов, которые установлены в маске. Ни один из операндов при этом не изменяется, но коды условий устанавливаются согласно результату логической операции И над двумя операндами.

Переход по значению одного бита. Вычислительный комплекс СМ1700 имеет ряд инструкций (табл. 4.3), которые опрашивают состояние одного бита или опрашивают и модифицируют бит

Таблица 4.3. **Инструкции перехода по значению одного бита**

Наименование	Мнемоника
Переход, если младший бит сброшен	BLBC S, A
Переход, если младший бит установлен	BLBS S, A
Переход, если бит установлен	BBS P, B, A
Переход, если бит сброшен	BBC P, B, A
Переход, если бит установлен, и безусловно установить бит	BBSS P, B, A
Переход, если бит установлен, и безусловно установить бит с блокировкой памяти	BBSSI P, B, A
Переход, если бит установлен, и безусловно сбросить бит	BBSC P, B, A
Переход, если бит сброшен, и безусловно установить бит	BBCS P, B, A
Переход, если бит сброшен, и безусловно сбросить бит	BBCC P, B, A
Переход, если бит сброшен, и безусловно сбросить бит с блокировкой памяти	BBCCI P, B, A

Примечание. P позиция бита, B базовый адрес, A адрес перехода, S операнд-источник.

в течение одной непрерываемой операции в оперативной памяти. Эта вторая форма особенно полезна для синхронизации нескольких процессов.

Специальным случаем инструкции BIT является пара инструкций перехода по младшему биту BLBC и BLBS, которые опрашивают младший (нулевой) бит двойного слова и выполняют переход, если он сброшен (BLBC) или установлен (BLBS). Эти инструкции предназначены для проверки результатов, возвращаемых подпрограммами или процедурами. В операционной системе МОС ВП, например, все системные подпрограммы и процедуры возвращают определенный код в регистр R0, давая индикацию конечного состояния. Значения этих кодов подобраны так, чтобы все коды ошибок были четными, а все успешные коды — нечетными. Таким образом можно легко узнать, успешно или ошибочно завершены процедуры (подпрограммы) с помощью инструкций BLBC и BLBS.

Инструкции переходов по значению одного бита (BBS и BBC) позволяют опрашивать и устанавливать (или сбрасывать) бит внутри произвольного битового поля. В любом случае единичный бит определяется позицией P и базой B опрашиваемого операнда. Если он находится в состоянии, соответствующем опросу, то выполняется переход по адресу A к инструкции, принимающей управление.

Для переходов с модификацией бита если переход удовлетворяется, то опрашиваемый бит устанавливается (BBSS и BBCS) или сбрасывается (BBSC и BBCC) независимо от начального состояния бита. Следует заметить, что рассмотренные инструкции перехода по биту используют позицию бита, в то время как инструкция BIT использует маску. Например, следующие два фрагмента эквивалентны, поскольку оба проверяют значение третьего бита в регистре R1 (нумерация битов с 0), и если он равен нулю, то выполняется переход:

```
BBC     #3,R1,NOTSET          BITB    #8,R1
                              BEQL    NOTSET
```

```
FLAG:    .LONG   1                ;Флажок первоначально установлен
                                  ;(свободно)


;
;<Программный код для первого процесса>
;
;Начало программы монопольногодоступа для первого процесса
;
WAIT1:   BBCC    #0,FLAG,WAIT1   ;Цикл до тех пор, пока флажок=1
         .
         <доступ к разделяемому ресурсу>
         .
         BISL    #1,FLAG         ;Восстановить флажок,
                                 ;продолжить работу


;
;<Программный код  для второго процесса>
;
;Начало программы монопольного доступа для второго процесса
;
WAIT2:   BBCC    #0,FLAG,WAIT2   ;Цикл до тех пор, пока флажок=1
         .
         <доступ к разделяемому ресурсу>
         .
         BISL    #1,FLAG         ;Восстановить флажок,
                                 ;продолжить работу
```

Рис. 4.1. Синхронизация двух процессов

Еще две инструкции BBSSI и BBCCI идентичны инструкциям BBSS и BBCC, за исключением того, что опрос и модификация бита выполняются с блокировкой памяти, т. е. если более чем один процессор или устройство прямого доступа одновременно обращаются к памяти, то только одному из них в это время будет разрешено опрашивать и модифицировать данный бит с гарантией непрерывности операции.

Инструкции переходов по биту могут использоваться для реализации семафорных операций P и V, предложенных Дейкстрой [1] и используемых практически в любой операционной системе. Следующий простой пример демонстрирует, как битовые инструкции могут использоваться для синхронизации двух процессов, протекающих в вычислительной машине. Как правило, два процесса разделяют какой-то ресурс так, что только один процесс может иметь доступ к ресурсу в данное время. Состояние ресурса определяет битовый флажок, разделяемый между процессами. Когда этот флажок установлен, ресурс свободен. Перед доступом к разделяемому ресурсу каждый процесс опрашивает этот бит и переходит к программе работы с ресурсом, если флажок установлен. Если флажок не установлен, то процесс ожидает (зацикливается на инструкции опроса) до тех пор, пока бит не станет равен единице. Когда процесс найдет бит установленным, он сбросит бит еще до того, как начнет пользоваться полученным ресурсом, и флажковый бит будет

```
;
; Процедура "SIEVE" для выработки простых чисел
;
;       SIZE  =  10000        ;Диапазон чисел от 0 до 10000
PRIMES:  .BLKB   SIZE/8       ;Создать первоначальную (обнуленную)
                              ;битовую таблицу PRIMES
SIEVE:                        ;Входная точка подпрограммы
        .WORD   ^M<>          ;Регистры не сохранять
        MOVL    #2,R1         ;Начать с простого числа 2
10$:    OUTASC  R1            ;Вывод простого числа (макрос)
        MOVL    R1,R0         ;Использовать R0 для формирования
                              ;кратных
;
; Установить все биты, которые кратны найденному простому числу
;
20$:    BBCS    R0,PRIMES,30$     ;Ликвидировать следующее кратное
30$:    ACBL    SIZE-1,R1,R0,20$  ;Формировать следующее кратное
;
; Поиск следующего нулевого бита в строке
;

40$:    BBC     R1,PRIMES,10$  ;Проверка на следующее простое число
        AOBLSS  SIZE,R1,40$    ;Продолжить просмотр битов
        RET                    ;Возврат в вызывавшую программу
```

Рис. 4.2. Процедура выработки простых чисел

сброшенным до тех пор, пока процесс не завершит работу с ресурсом. Такая координация процессов может быть запрограммирована так, как показано на рис. 4.1. Здесь оба процесса опрашивают флажок. Один из процессов как только обнаружит, что флажок установлен, получит доступ к основному телу его программного кода. Другой процесс продолжает сбрасывать уже сброшенный бит флажка до тех пор, пока флажок не установится в единицу первым процессом.

Следует заметить, что во всех инструкциях работы с одиночными битами и битовыми полями переменной длины операнд «позиция бита» интерпретируется как двойное слово знакового смещения от базового адреса, т. е. выбранный бит может находиться в пределах от -2^{31} до $2^{31}-1$ бит от младшего (нулевого) бита базового адреса. Таким образом, старшие 29 бит операнда «позиция» содержат знаковое смещение в байтах от базового адреса до выбранного байта, а три младших бита операнда «позиция» — номер бита внутри выбранного байта.

Хорошей иллюстрацией использования такой возможности является показанная на рис. 4.2 процедура SIEVE выработки простых чисел. Она использует строку из N бит для представления целых чисел от 0 до $N-1$, где позиция каждого бита соответствует целому числу (например, бит 6 в строке представляет целое число 6). Процедура начинает с простого числа 2 и выделяет все числа, кратные 2, установкой битов, номера позиций которых кратны 2. Они не могут быть простыми, поскольку точно делятся на 2. Затем процедура ищет следующий нулевой бит в строке. Этот бит представляет простое число, так как он не кратен любому меньшему числу. Как только простое число

Таблица 4.4. **Инструкции обработки битовых полей**

Наименование	Мнемоника	Алгоритм
Чтение поля	EXTV P, L, B, D	Пересылает указанное поле в операнд D к его правому краю и распространяет знак поля
Чтение поля с расширением нулями	EXTZV P, L, B, D	Пересылает указанное поле в операнд D к его правому краю и распространяет нули в старших разрядах
Сравнение поля	CMPV P, L, B, S	Сравнивает знаково-расширенное поле с операндом S, устанавливая коды условий
Сравнение поля с расширением нулями	CMPZV P, L, B, S	Поле, расширенное нулями, сравнивается с операндом S, и устанавливаются коды условий
Запись поля	INSV S, P, L, B	Операнд L определяет число младших битов, перемещаемых из операнда S в указанное поле
Поиск первого установленного бита	FFS P, L, B, F	Определяет местоположение младшего единичного бита в указанном поле
Поиск первого сброшенного бита	FFC P, L, B, F	Определяет местоположение младшего нулевого бита в указанном поле

Примечание. P позиция начального бита, L размер поля, B базовый адрес, S операнд-источник, D операнд-приемник, F найденная позиция бита.

найдено, все полученные к этому моменту кратные отделяются и продолжается поиск следующего нулевого бита.

Битовые поля переменной длины. Битовое поле описывается тремя аргументами: его базовым адресом, позицией первого бита относительно базы и размером поля в битах. Инструкции работы с битовыми полями приведены в табл. 4.4.

При чтении битового поля оно перемещается к правому краю внутри двухсловного операнда-приемника. Старшие биты приемника могут быть либо обнулены, либо «расширены» знаком поля в зависимости от инструкции.

Примером использования инструкции EXTZV является печать шестнадцатеричного значения 32-разрядного удвоенного слова (рис. 4.3). Эта инструкция используется внутри цикла для получения слева направо восьми четырехразрядных цифр. При этом макрос OUTCHAR вызывает печать цифры, содержащейся в R3. При каждом выполнении инструкции EXTZV четырехразрядное поле извлекается из R1 и помещается в крайних правых позициях регистра R3 с распространением нулей в оставшихся позициях. Регистр R2 указывает начальную позицию бита в строке.

Инструкции FFS и FFC используются для определения местоположения первого установленного или сброшенного бита

```
; Печать удвоенного слова в шестнадцатеричном символьном коде
;
;       R1 - содержит вычисляемое двойное слово
;       R2 - позиция четырехразрядного поля,
;            которое должно быть извлечено
;
DIGITS:         .ASCII   /0123456789ABCDEF/ ;Символьное обозначение
PRINT:
        MOVL    NUMBER,R1       ;Исходное двойное слово,
        MOVL    #<<4*8>-4>,R2   ;Начать с последней
                                ;шестнадцатеричной цифры
10$:    EXTZV   R2,#4,R1,R3     ;В R3 следующая двойная цифра
        MOVB    DIGITS[R3],R3   ;В R3 цифра в коде СМ1700
        OUTCHAR R3              ;Вывод шестнадцатеричной цифры
        ACBL    #0,#-4,R2,10$   ;Продолжить слева направо
```

Рис. 4.3. Печать шестнадцатеричного двойного слова

в поле размером от 0 до 32 бит. Если бит не найден в указанном поле, то эти инструкции вырабатывают позицию следующего за указанным полем бита и устанавливают бит условия Z (в противном случае Z сброшен). Например, инструкция FFS #0, #32, FIELD, R0 в зависимости от содержимого ячейки FIELD будет устанавливать в R0 и бите Z следующие значения:

FIELD	R0	Z
1	0	0
4	2	0
8	3	0
9	0	0
0	32	1

Как было замечено в предыдущем разделе, операнд P начальной позиции может быть больше, чем 32 бита относительно базового адреса. Это оказывается полезным при просмотре длинных полей. За одну инструкцию в поле могут быть просмотрены 32 бита, и если указанный бит не найден, то операнд F (найденная позиция) будет содержать начальную позицию следующего 32-разрядного поля для поиска.

4.3. СИМВОЛЬНЫЕ СТРОКИ

В СМ1700 существует функционально полный набор инструкций, обрабатывающих символьные строки. Сюда относятся инструкции пересылки, сравнения и поиска символьных строк длиной до 65 536 байт. В описаниях этих инструкций термины «байт» и «символ» эквивалентны, поскольку эти инструкции имеют дело со строками байтов независимо от используемого представления символов. Тем не менее они называются инструкциями обработки символьных строк, потому что обычно применяются для символьных текстовых строк.

Символьная строка определяется длиной строки в байтах (длина содержится в одном слове, поскольку максимальная строка содержит 64 Кбайт) и адресом самого первого символа

строки. Инструкции символьных строк используют от двух до шести общих регистров (из числа R0 ... R5) для хранения промежуточных значений и для выработки продвинутых указателей строки. Рекомендуется использовать инструкции PUSHR и POPR для сохранения и восстановления регистров, которые модифицируются при обработке символьных строк.

Например, рассмотрим одну из инструкций MOVC3 (пересылка). Формат инструкции следующий: MOVC3 L, A_S, A_D (где L — длина строки-источника, A_S — адрес строки-источника, A_D — адрес строки-приемника). Инструкция MOVC3 пересылает строку символов из одного места в другое, т. е. копирует блок памяти. После выполнения инструкции MOVC3 регистры будут иметь следующие значения: R0, R2, R4, R5 = 0; R1 — адрес байта, располагающегося вслед за строкой-источником; R3 — адрес байта, располагающегося вслед за строкой-приемником.

Ниже приведен пример, в котором имеются три строки сообщений MS1 ... MS3, которые будут пересылаться в буфер вывода BUFFER в зависимости от значения, содержащегося в регистре R7. Каждая из этих строк запомнена как символьная строка со счетчиком (директива .ASCIC), т. е. первый байт содержит длину строки (сюда не входит сам байт-счетчик). Если регистр R7 содержит ноль, будет пересылаться первое сообщение, если R7 содержит единицу — второе сообщение и т. д.:

```
          MAXMS=2                    ;Максимальный номер сообщения
ADDRS:    .ADDRESS MS1,MS2,MS3       ;Адреса сообщений
MS1:      .ASCIC   /сообщение 1/
MS2:      .ASCIC   /сообщение 2/
MS3:      .ASCIC   /сообщение 3/
BUFFER:.BLKB       100                ;100-байтный буфер вывода

          CMPL     R7,#MAXMS         ;Оценить правильность
                                     ;номера сообщения
          BGTRU    ERROR             ;Переход, если номер
                                     ;слишком большой
          MOVL     ADDRS[R7],R9      ;Получить сообщения
          MOVZBL   (R9)+,R8          ;R8=длина сообщения,
                                     ;R9=адрес первого символа
          MOVC3    R8,(R9),BUFFER    ;Переслать строку в буфер вывода
```

Инструкция MOVC5, так же как и инструкция MOVC3, копирует блок памяти, но строки источника и приемника могут быть различны по длине. Если источник длиннее, чем приемник, то копируется столько символов, сколько позволяет длина приемника (т. е. строка усекается). Если источник короче, чем приемник, то остаток приемника заполняется копиями символа-наполнителя, указанного в качестве одного из операндов. Формат инструкции MOVC5 следующий: MOVC5 L_S, A_S, F, L_D, A_D (где L_S — длина источика, A_S — адрес источника, F — наполнитель, L_D — длина приемника, A_D — адрес приемника).

После выполнения инструкции MOVC5 состояние регистров такое же, как в случае выполнения MOVC3, за исключением

```
;  ВХОДНЫЕ ПАРАМЕТРЫ:
        BUFLEN(AP) - длина проверяемого буфера
        BUFADR(AP) - адрес буфера
;  ВЫХОДНЫЕ ПАРАМЕТРЫ:
        RETKEY(AP) - если имя правильное, то указанный адрес содержит
                     номер фигуры: 0 = король, 1 = ферзь, 2 = слон,
                     3 = конь, 4 = ладья, 5 = пешка.
;  ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ:
    R0=0 если нет совпадения, =1 если правильное имя найдено
;
        BUFLEN =4               ;Смещение до аргумента длины
        BUFADR =8               ;Смещение до аргумента адреса
        RETKEY =12              ;Смещение до адреса результата
;
;  Список строк имен для шахматных фигур. Нулевой байт
;  ограничивает список.
;
NAME:   .ASCIC  /KING/          ;Король
        .ASCIC  /QUEEN/         ;Ферзь
        .ASCIC  /BISHOP/        ;Слон
        .ASCIC  /KNIGHT/        ;Конь
        .ASCIC  /ROOK/          ;Ладья
        .ASCIC  /PAWN/          ;Пешка
        .BYTE   0               ;Ограничитель списка

VDATE:  .WORD   ^M<R2,R3,R4,R5,R6,R7>   ;Сохраняемые регистры
        SKPC    #^A/ /,BUFLEN(AP),@BUFADR(AP) ;Пропуск пробелов
        BEQL    30$                     ;Выход, если ничего нет
        MOVQ    R0,R5                   ;Сохранить длину/адрес в R5/R6
        CLRL    R7                      ;Обнулить индикатор фигуры
        MOVAL   NAME,R1                 ;Получить адрес списка фигур
10$:    MOVZBL  (R1)+,R0                ;R0=длина имени фигуры
        BEQL    30$                     ;Выход, если таблица исчерпана
        CMPC5   R0,(R1),#^A/ /,R5,(R6)  ;Строки совпадают?
        BEQL    20$                     ;Выход, если строки совпадают
        ADDL    R0,R1                   ;R1=адрес следующего имени
        INCL    R7                      ;Следующий номер фигуры
        BRB     10$                     ;Продолжить цикл
20$:    MOVZBL  #1,R0                   ;Код успешного завершения
        MOVL    R7,@RETKEY(AP)          ;Вернуть номер фигуры
        RET
30$:    CLRL    R0                      ;Код ошибочного завершения
        RET
```

Рис. 4.4. Определение имени шахматной фигуры

того, что R0 содержит число непересланных символов, если источник был длиннее, чем приемник. Кроме того, коды условий устанавливаются на основе сравнения длин источника и приемника. Инструкция MOVC5 с нулевой длиной источника может быть использована для заполнения блока памяти кодом какого-то одного символа, указанного в качестве наполнителя. Например, инструкция MOVC5 #0, (R0), #0, #512, OUTBUF заставляет обнулиться 512-байтный буфер OUTBUF. Заметим, что операнд-источник, указанный посредством R0, на самом деле не существует и, следовательно, нет необходимости, чтобы R0 содержал правильный адрес.

Полный перечень инструкций обработки строк приведен в приложении 1, а ниже описаны некоторые из них (при описании использованы следующие обозначения: L — длина строки, A —

адрес строки, F — наполнитель, C — символ, S — источник, D — приемник).

Сравнение двух строк. Имеются две модификации: для трех операндов (CMPC3 L, A_S, A_D) и для пяти операндов (CMPC5 L_S, A_S, F, L_D, A_D). Выполняется побайтное сравнение и устанавливаются коды условий по первым несовпавшим байтам, если они вообще есть.

Локализация символа: LOCC C, L_S, A_S. После этой инструкции R1 содержит либо адрес найденного символа, либо адрес байта, непосредственно следующего за строкой, если символ не был найден. Регистр R0 содержит число оставшихся символов в строке.

Пропуск символов: SKPC C, L_S, A_S. Эта инструкция подобна LOCC, за исключением того, что выполняется проверка на неравенство вместо равенства.

Поиск подстроки символов: MATCHC L_S, A_S, L_D, A_D. Строка S, указанная операндами L_S и A_S, ищется как подстрока, входящая в строку D, указанную операндами L_D и A_D. Если подстрока найдена, то R0 содержит число байтов, оставшихся в строке D, R1 — адрес найденной подстроки. Если подстрока не найдена, то R0 содержит все нули, R1 указывает на байт, следующий после строки D.

В качестве примера использования инструкций символьных строк на рис. 4.4 приведена процедура CHESS, которая проверяет входной буфер, чтобы определить, не содержит ли он имени шахматной фигуры. Начальные пустые символы первыми удаляются из входного буфера, и затем строка проверяется на имена шахматных фигур, запомненных в массиве NAME. Если имя найдено, то возвращенная (из процедуры) переменная указывает, какая фигура была найдена. При возврате регистр R0 указывает, было ли вообще найдено имя.

4.4. ДЕСЯТИЧНАЯ АРИФМЕТИКА

Для некоторых языков, подобных Коболу, часто удобнее трактовать ЭВМ как десятичную машину, а не как двоичную. Поэтому СМ1700 имеет аппаратно реализованную десятичную и двоичную арифметику. Внутренним представлением числа для десятичной арифметики является упакованная десятичная строка. Эта строка есть непрерывная последовательность 4-разрядных кодов, каждый из которых представляет десятичную цифру от 0 до 9, причем в младшей тетраде представлен знак числа. Десятичная строка описывается ее длиной в цифрах (число тетрад) и адресом младшего значащего байта. Ассемблерная директива .PACKED используется для запоминания одной упакованной строки переменной длины до 31-й десятичной цифры в последовательных байтах.

Аналогично набору инструкций двоичной арифметики набор десятичной арифметики включает инструкции пересылки, сравнения, сложения и др. (см. табл. 4.5). Эти инструкции всегда

Таблица 4.5. **Инструкции десятичной арифметики**

Выполняемое действие	Мнемоника
Пересылка строки источника в строку приемника	MOVP L, A_S, A_D
Сравнение строки источника со строкой приемника	COMP L, A_S, A_D
Сложение строки источника со строкой приемника	ADDP L_S, A_S, L_D, A_D
Вычитание строки источника из строки приемника	SUBP L_S, A_S, L_D, A_D
Умножение двух сомножителей (операнды 1 и 2) с запоминанием результата в третьем операнде	MULP L_1, A_1, L_2, A_2, L_3, A_3
Деление второго операнда на первый и запоминание результата в третьем операнде	DIVP L_1, A_1, L_2, A_2, L_3, A_3
Преобразование десятичной строки в целое число удвоенной длины	CVTPL L_S, A_S, A_D
Преобразование целого числа удвоенной длины в упакованное десятичное число	CVTLP A_S, L_D, A_D

Примечание. L длина строки; A адрес строки; S источник; D приемник; 1, 2, 3 номера операндов.

трактуют десятичные строки как целые числа с десятичной запятой справа от младшей значащей цифры строки. Если результат десятичной инструкции должен быть запомнен в строке, которая больше, чем результат, то старшие тетрады заполняются нулями. Таким образом, единственная разница между этими инструкциями и инструкциями двоичной арифметики заключается в том, что требуется операнд, определяющий размер строки, поскольку упакованные десятичные строки могут быть разной длины.

Установка кодов условий для десятичной арифметики подобна инструкциям двоичной арифметики (но с десятичным переполнением результата), если строка приемника слишком коротка, чтобы содержать все ненулевые цифры результата. Если переполнение возникает, строка приемника замещается правильными младшими значащими цифрами результата со знаком.

В СМ1700 имеется также набор инструкций для преобразования из упакованного десятичного формата в другие форматы. Таким образом, представление числа может быть выбрано для конкретного применения и преобразовано, если необходимо, в другое внутреннее представление для последующей обработки. Как правило, инструкции для десятичных строк используются компиляторами языков программирования и редко используются на уровне ассемблера.

4.5. АРИФМЕТИКА С ПЛАВАЮЩЕЙ ТОЧКОЙ

Логические и арифметические инструкции для целых чисел наглядно демонстрируют, каким образом коды операций, типы данных и режимы адресации могут сочетаться в одной инструкции. Большинство инструкций, предназначенных для целых чисел, используются также для чисел с плавающей точкой,

представленных в форматах F, D, H и G. Существует также полный набор инструкций преобразования между форматами данных. Таблица 4.6 содержит некоторые инструкции для арифметики с плавающей точкой в форматах одинарной (F) и двойной (D) точности. Полный перечень инструкций приведен в приложении 1.

Инструкции для арифметики с плавающей точкой включают как двух-, так и трехоперандную формы. Для иллюстрации рассмотрим следующий оператор языка Фортран:

```
A(I) = B(I) * C(I)
```

Таблица 4.6. **Инструкции арифметики с плавающей точкой**

Выполняемое действие	Мнемоника
Пересылка первого операнда O_1 на место второго операнда O_2	MOV{F,D} O_1,O_2
Очистка операнда O_1	CLR{F,D} O_1
Преобразование операнда O_1 одинарной точности в операнд O_2 двойной точности	CVTFD O_1,O_2
Преобразование операнда O_1 (B, W, L)-типа в операнд O_2 (F, D)-типа	CVT{B,W,L} {F,D} O_1,O_2
Преобразование операнда O_1 (F, D)-типа в операнд O_2 (B, W, L)-типа	CVT{F,D} {B,W,L} O_1,O_2
Сравнение двух операндов	CMP{F,D} O_1,O_2
Тест операнда	TST{F,D} O_1,O_2
Сложение двух операндов и запись результата на место второго или третьего операнда	ADD{F,D} {2,3} O_1,O_2,O_3
Вычитание операнда O_1 из O_2 и запись результата на место O_2 или O_3	SUB{F,D} {2,3} O_1,O_2,O_3
Умножение O_1 и O_2 и запись результата на место O_2 или O_3	MUL{F,D} {2,3} O_1,O_2,O_3
Деление O_2 на O_1 и запись результата на место O_2 или O_3	DIV{F,D} {2,3} O_1,O_2,O_3
Расширенное умножение	EMOD{F,D} O_1,O_2,O_3,O_4,O_5
Вычисление полинома	POLY{F,D} O_1,O_2,O_3

где A, B и C являются статическими массивами вещественных чисел формата REAL*4, а I — целочисленная переменная формата INTEGER*4. Последовательность инструкций, реализующих эту операцию, следующая:

```
MOVL    I,R0
MULF3   B[R0],C[R0],A[R0]
```

Та же последовательность инструкций применяется, если массивы используют типы данных REAL*8, INTEGER*4, INTEGER*2 или INTEGER*1. В этом случае инструкция MULF3 заменяется на MULD3, MULL3, MULW3 или MULB3 соответственно.

Инструкция EMOD умножает число с плавающей точкой O_3 на число с плавающей точкой расширенной точности O_1O_2 (операнд O_1 расширяется операндом O_2 до 9 или 19 цифр точности для форматов F или D соответственно) и выдает ответ в виде целой части O_4 и дробной O_5, каждая из которых представляется отдельным числом в соответствующем формате. Такая инструкция может быть полезна для формирования аргумента тригонометрических и экспоненциальных функций через определенный интервал.

Интересной инструкцией является POLY, которая вычисляет полином по таблице коэффициентов, используя метод Горнера [2]. Результат формируется в R1—R0 (для POLYF) или в R3—R0 (для POLYD). Имея вещественную переменную X и таблицу PTABL, содержащую N чисел с плавающей точкой C_0, C_1, ..., C_N, можно вычислить полином $P=C_0+X\cdot C_1+X^2\times \times C_2+...+X^N\cdot C_N$ следующим образом:

```
        POLYF   X,#N,PTABL
          .
          .
PTABL:  .FLOAT  <Константа N>   ;C(N)
        .FLOAT  <Константа N-1> ;C(N-1)
          .
          .
        .FLOAT  <Константа 0>   ;C(0)
```

Инструкции EMOD и POLY могут быть полезны для решения различных тригонометрических, экспоненциальных и полиномиальных уравнений.

4.6. МНОГОЭЛЕМЕНТНЫЕ СТРУКТУРЫ

Массивы, списки, очереди и деревья являются типичными структурами данных, обычно используемых как в языках высокого уровня, так и в ассемблерах. Эти структуры данных образуют однородные объединения элементов данных, которые называются записями, а объединения записей — файлами. Каждая запись является в действительности логическим элементом данных с его собственными характеристиками (размером, типом и начальным

значением), группируемыми по определенным правилам с целью образования сложной структуры данных. Записи и файлы являются логическими единицами, которые должны быть обработаны на основе знания их структуры. Следовательно, необходимы различные инструкции и режимы адресации для доступа к элементам различных записей.

Массивы. Простейшей реализацией массива является непрерывное объединение одномерных и одинаковых по размеру элементов памяти. В таком виде доступ к любому отдельному элементу выполняется вычислением адреса в зависимости от размера (байт, слово, двойное слово и т. д.) и позиции (индекса) элемента в массиве. Например, для вычисления оператора J=A*K+B(I) программа на языке ассемблера может быть следующей:

```
MOVL    I,R2        ;Загружить индекс в регистр
CVTLF   K,R1        ;Преобразовать целое К в вещественное
MULF2   A,R1        ;Сформировать A*K как вещественную величину
ADDR2   B[R2],R1    ;Прибавить B, чтобы сформировать A*K+B(I)
CVTFL   R1,J        ;Запомнить целое (результат) в ячейку J
```

Когда используются многомерные массивы, проблема адресации конкретного элемента становится частью задачи размещения в памяти ЭВМ многомерного массива в виде линейного (одномерного). Как правило, массивы запоминаются по столбцам. Например, двумерный массив A разместится в памяти с помощью фортран-компилятора следующим образом:

```
A(0,0)  A(0,1)
                .......> A(1,1)  A(0,1)  A(1,0)  A(0,0)
A(1,0)  A(1,1)
```

Здесь A(0, 0) располагается по младшему адресу, а A(1, 1) — по старшему. Для того чтобы добраться до элемента A(I, J), необходимо вычислить индекс столбца, а затем элемент внутри столбца. При этом требуется информация о границах массива.

Чтобы облегчить компиляторам адресацию массивов, в СМ1700 предусмотрена инструкция INDEX (индекс массива). Она вычисляет индекс для одномерного массива из элементов данных с одинаковой длиной как целых, так и вещественных типов, а также для массивов из битовых полей, символьных строк и десятичных строк. Она также проверяет нижние и верхние границы индекса для заданного массива. Формат инструкции следующий: INDEX B, L, H, S, $I_{вх}$, $I_{вых}$, где B — базовый индекс, L — нижний предел, H — верхний предел, S — размер, $I_{вх}$ — входной индекс, $I_{вых}$ — выходной индекс.

Операнд $I_{вх}$ складывается с операндом B, сумма умножается на операнд S, и результат записывается на место операнда $I_{вых}$. Если при этом B<L или B>H, то происходит внутреннее прерывание при нарушении индексного диапазона.

Ниже приводится несколько примеров использования инструкции INDEX (совместно с другими инструкциями) для работы с массивами.

```
DECLARE A(-3:10)   BIT(5);
        A(I) = 1;
```

компилируются в программу на языке ассемблера

```
INDEX   I,#-3,#10,#5,R0
INSV    #1,R0,#5,A
```

Следующие операторы языка Фортран

```
INTEGER*4       A(L1:N1,L2:N2),I,J
A(I,J)=1
```

могут быть представлены на языке ассемблера:

```
INDEX   J,#L2,#N2,#M1,#0,#R0     ;где M1=N1-L1+1
INDEX   I,#L1,#N1,#1,R0,R0
MOVL    #1,A-K[R0]               ;где K=((N*M1)+L1)*4
```

Наличие операнда входного индекса $I_{вх}$ обеспечивает последовательное использование инструкции INDEX для многомерных массивов. Для одномерных массивов полей переменной длины при вычислении индекса он позволяет вводить постоянную составляющую, что не обеспечивается обычной адресной арифметикой.

Циклические списки. Одно из общепринятых применений списка состоит в реализации циклической очереди. Циклический список, показанный на рис. 4.5, используется для передачи информации между двумя или более процедурами программы. Одна процедура является обычно производителем данных, а другая — потребителем. Первая передает информацию в список, а другая берет оттуда информацию и обрабатывает ее.

Два указателя «Верх» и «Низ» указывают соответственно на следующий удаляемый элемент и на следующее место для добавления данных. Когда процедура-производитель создает данные, она помещает их на место, определяемое указателем «Низ»,

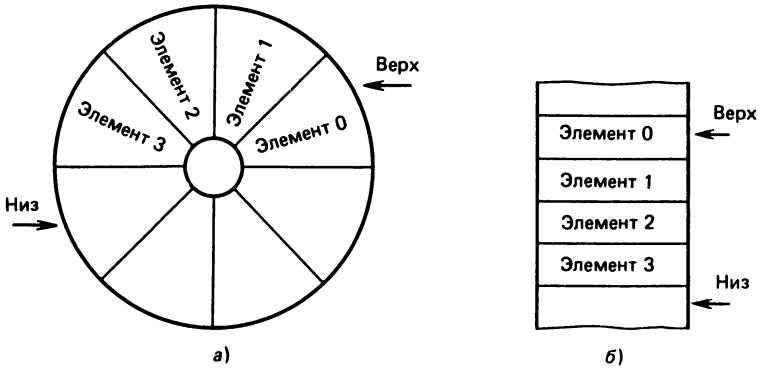


Рис. 4.5. Циклический список:
а — логическая структура; *б* — физическая структура в памяти

и продвигает указатель, чтобы указать на следующий элемент. Когда процедуре-потребителю нужно получить данные, она выбирает элемент, определяемый указателем «Верх», и корректирует его, чтобы указать на следующий элемент. Поскольку циклический буфер реализуется в линейной памяти, процедуры проверяют конец списка, когда корректируют указатели. Если указатель вышел за конец списка, то он сбрасывается, чтобы указать на первый элемент списка.

Существует одно ограничение: процедура-производитель не должна добавлять элементы в список быстрее, чем процедура-потребитель способна выбирать их. В противном случае буфер переполняется и данные теряются.

На рис. 4.6 приведены процедуры INSBUF и REMBUF для добавления и удаления символов циклического буфера. Такие процедуры могут быть использованы программой обслуживания терминалов для того, чтобы получить полные строки от удаленного терминала и передать их интерпретатору команд операционной системы для обработки.

Инструкция EDIV, используемая в этих процедурах для реализации модульной функции, автоматически перемещает указатель назад по кругу к началу списка, если он достигнет конца списка. Это происходит всякий раз, когда указатель становится равным BUFSIZE; при этом остаток от деления равен нулю. В противном случае остаток такой же, как исходный индекс.

Однократно связные списки. Связанные структуры данных могут рассматриваться как массивы, в которых последовательные элементы занимают произвольные (непоследовательные) ячейки памяти. Чтобы найти элемент N массива или перейти от элемента N к элементу N+1, необходимо иметь указатель этого элемента, а именно его адрес. В простейшей схеме (рис. 4.7) имеется список указателей фиксированной длины, которые определяют место каждого элемента в структуре данных. Элементы могут быть фиксированной или переменной длины. Ноль в одном из элементов списка указателей показывает, что с ним не связан никакой элемент данных.

При такой организации структуры данных фактически добавляется уровень косвенности для адресации массива. Это позволяет избежать ограничения на длину элементов данных и их непрерывность в памяти. Размещение самого списка указателей должно быть фиксированным; кроме того, необходимо знать максимальное число размещаемых элементов. Таким образом, «цена» одного указателя составляет 4 байта на элемент (размер элемента может быть любым).

Один из способов облегчить проблему перераспределения списка указателей состоит в том, чтобы включить указатель внутрь каждого элемента данных (рис. 4.8). Заголовок списка

необходим для определения местонахождения первого элемента данных, который затем указывает на следующий элемент и т. д.

```
        .SBTTL  -  ПРОГРАММА УПРАВЛЕНИЯ ЦИКЛИЧЕСКИМ БУФЕРОМ
;
;Память для буфера и переменные для указателей.
;
                BUFSIZE=100             ;Размер буфера
        BUFFER: .BLKB   BUFSIZE         ;Кольцевой буфер символов
        COUNT:  .LONG   0               ;Число элементов в буфере
        BOTTOM: .LONG   0               ;Индекс последнего (нижнего)
                                        ;элемента списка
        TOP:    .LONG   0               ;Индекс первого (верхнего)
                                        ;элемента списка

;++      INSTBUF - ПРОЦЕДУРА ДОБАВЛЕНИЯ В ЦИКЛИЧЕСКИЙ БУФЕР
;
;Входные параметры:
;       CHAR(AP) - младший байт первого аргумента
                   содержит добавляемый символ
;
;Возвращаемые значения:
;      R1=0 - буфер полный, 1 - символ успешно добавлен
;--
                CHAR=4                  ;Смещение
INSBUF: .WORD   ^M<>                    ;Нет сохраняемых регистров
        CLRQ    R0                      ;Считать полный буфер ошибкой
        CMPL    #BUFSIZE,COUNT          ;Буфер заполнен?
        BEQL    10$                     ;Выход по ошибке, если это так
        MOVL    BOTTOM,R1               ;Индекс последнего элемента
        MOVB    CHAR(AP),BUFFER[R1]     ;Добавить символ в буфер
        INCL    COUNT                   ;Учесть добавление символа
        INCL    R1                      ;Скорректировать индекс
        EDIV    #BUFSIZE,R1,R0,BOTTOM   ;"Провернуть" по кругу
                                        ;указатель BOTTOM
        MOVZBL  #1,R1                   ;Код успешного завершения
10$:    RET                             ;Возврат

;++       REMBUF - ПРОЦЕДУРА УДАЛЕНИЯ ИЗ ЦИКЛИЧЕСКОГО БУФЕРА
;
;Входные параметры:
;       CHARADR(AP)-адрес байта для приема извлекаемого символа
;
;Возвращаемые значения:
;       R1=0 - буфер пуст, 1 - символ успешно извлечен
;--
        CHARADR=4                       ;Смещение
REMBUF: .WORD   ^M<>                    ;Нет сохраняемых регистров
        CLRQ    R0                      ;Предполагаем буфер пустым
        TSTL    COUNT                   ;Буфер пуст?
        BEQL    10$                     ;Если да, то выход
        MOVL    TOP,R1                  ;Индекс верхнего элемента
        MOVB    BUFFER[R1],@CHARADR(AP) ;Вернуть символ
                                        ;вызвавшей программе
        DECL    COUNT                   ;Учесть удаление символа
        INCL    R1                      ;Скорректировать индекс
        EDIV    #BUFSIZE,R1,R0,TOP      ;"Провернуть" по кругу
                                        ;указатель TOP
        MOVZBL  #1,R1                   ;Код успешного завершения
10$:    RET                             ;Возврат
```

Рис. 4.6. Программа управления циклическим буфером

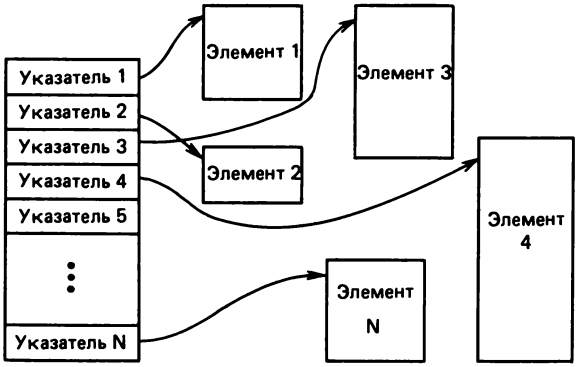


Рис. 4.7. Массив со списком указателей

Последний элемент данных содержит нулевой указатель, показывая, что далее не следует ни одного элемента, т. е. что это конец списка.

Такая структура, известная как однократно связный список, не позволяет прямо адресовать или определять место элемента с помощью номера индекса, потому что поиск необходимо вести через цепочку связей. Однако в однократно связном списке легко переходить от одного элемента к другому, добавлять и исключать элементы. На рис. 4.9 показаны связные списки до и после того, как элемент X вставлен за элементом A.

Ниже приведена последовательность инструкций, которые будут вставлять новый элемент в список, предполагая, что регистр R2 содержит адрес добавляемого элемента, а регистр R1 — адрес элемента, за которым следует новый элемент:

```
MOVL    (R1),(R2)       ;Связать новый элемент со следующими
MOVAL   (R2),(R1)       ;Связать предыдущий элемент с новым
```

Операция удаления элемента из списка еще короче (регистр R2 содержит адрес удаляемого элемента, а R1 — адрес предыдущего элемента):

```
MOVAL   (R2),(R1)       ;Связать предыдущий элемент с последующим
```

Использование однократно связного списка показательно при управлении пулом динамической памяти. (Под пулом понимается

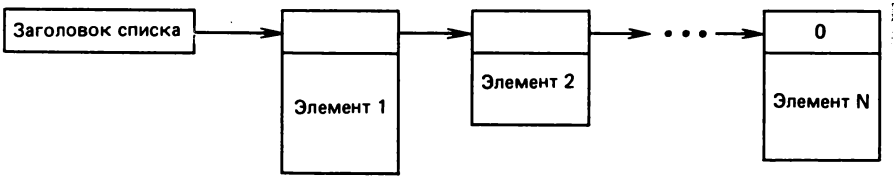


Рис. 4.8. Однократно связный список

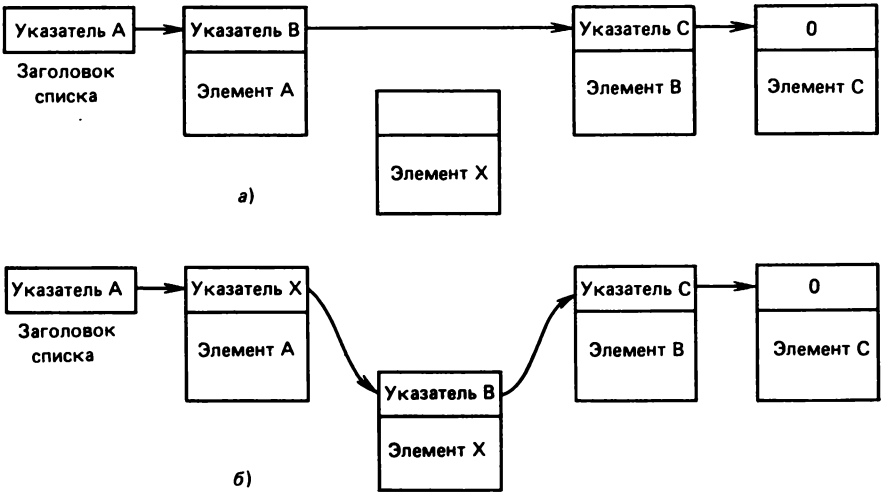


Рис. 4.9. Однократно связный список до вставки (*а*) и после вставки (*б*) элементов

область оперативной памяти, резервируется для размещения промежуточных переменных, констант, данных ввода-вывода и т. п.) В операционной системе один большой пул может разделяться многими программами и пользователями. Если программе необходимо разместить блок памяти для динамической структуры данных, она обращается к распределителю пула, требуя блок нужного размера. Когда программа заканчивает работу с блоком, он возвращается к распределителю пула. Динамическое распределение экономит объем памяти в системе, поскольку нет необходимости для каждой программы отводить статическую память, которая используется эпизодически.

Структура пула динамической памяти (рис. 4.10) используется в операционной системе МОС ВП. Свободный пул является списком блоков памяти. Первые два двойных слова каждого блока содержат размер блока и указатель на следующий блок. Когда система инициализируется, существует только один большой блок (рис. 4.10, *а*). Как только пользователи начинают занимать и возвращать участки памяти, пул становится раздробленным и содержит много прерывающихся блоков различных размеров (рис. 4.10, *б*). Если блок памяти, возвращенный в пул, примыкает к находящемуся здесь блоку, то эти два блока объединяются в один большой блок. Если все блоки возвращены, то список будет возвращен в исходное состояние (рис. 4.10, *а*). Для облегчения объединения блоков список хранится в порядке возрастания адресов памяти.

Процедура ALLOC (рис. 4.11, 4.12) отводит в пуле блок памяти переменного размера. Она просматривает список для

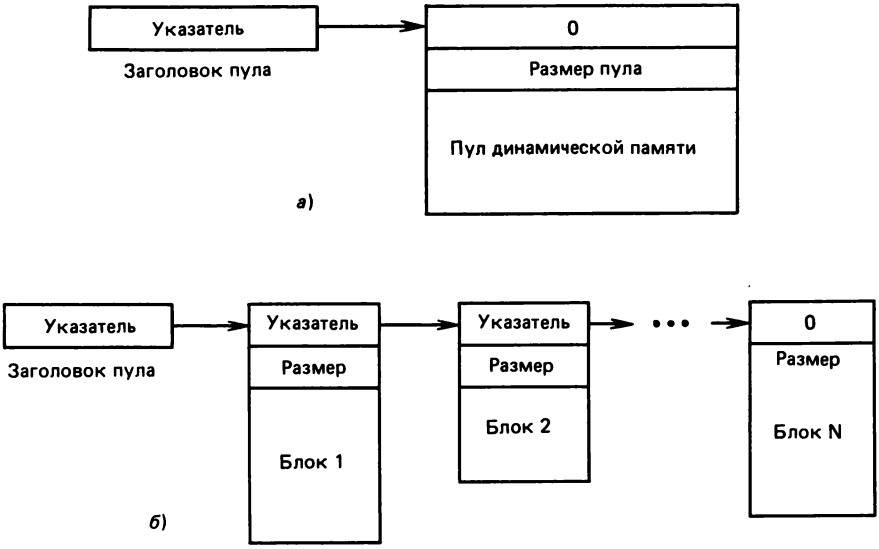


Рис. 4.10. Начальный (*а*) и фрагментированный (*б*) списки пула динамической памяти

поиска блока подходящего размера. Если она находит блок точно требуемого размера, то блок удаляется из цепочки и передается вызвавшей программе. Если же она находит блок большего размера, то блок разбивается на две части — одну требуемого размера и другую — остаток найденного большого блока. Запрошенный блок передается вызвавшей программе, а остаток с его новым размером возвращается в список.

Эта процедура демонстрирует эффективное использование адресации с автоувеличением, автоуменьшением и косвенной для обработки списка и соответствующих указателей.

Двукратно связные списки. Существуют некоторые задачи, которые требуют просмотра списка как назад, так и вперед или же вставки его в конец списка. Это возможно при наличии указателей на последний и на первый элементы списка, т. е. необходим двукратно связный список. Такой список в каждом элементе имеет указатель вперед и назад (рис. 4.13). Двукратно связный список часто используется для реализации очереди, т. е. линейной структуры, в которой элементы добавляются в конец, а удаляются из начала структуры.

В СМ1700 каждый элемент очереди описывается заголовком: парой слов удвоенной длины, где первое является связью вперед, а второе — связью назад. На рис. 4.14 показан процесс добавления элементов в очередь (заметим, что связи и вперед, и назад указывают на ячейку Н заголовка). Указатель вперед из последнего элемента всегда указывает на заголовок очереди.

```
        .SBTTL  ALLOC -- Процедура выделения памяти
;
;  ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ:
;
;    Выделить блок памяти из списка динамического пула.
;    (Запрашиваемые блоки не менее 8 байтов длиной.)
;
;  ВХОДНЫЕ ПАРАМЕТРЫ:
;    SIZE(AP) - размер требуемого блока
;    RETADR(AP) - адрес результата
;
;  ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ:
;       R1 = 0, если блок не найден; = 1, если блок выделен.
;       @RETADR(AP) - адрес выделенного блока
;
        SIZE=4                  ;Смещение до аргумента размера
        RETADR=8                ;Смещение до адреса результата

ALLOC:
        .WORD   ^M<R3,R4>       ;Сохранить регистры
        MOVAL   LISTHEAD,R1     ;Адрес заголовка списка памяти
        MOVL    SIZE(AP),R0     ;Размер запрашиваемого блока
10$:    MOVL    R1,R3           ;Адрес предыдущего блока
        MOVL    (R3),R1         ;Адрес следующего блока
        BEQL    30$             ;Выход по ошибке, если 0
        CMPL    R0,4(R1)        ;Этот блок достаточно большой?
        BGTRU   10$             ;Если нет, найти другой блок
        BEQL    20$             ;Переход, если найден блок
                                ;точно требуемого размера
;
;     Найден блок памяти, который больше, чем нужно. Выделить
;     требуемый размер и сформировать новый блок из остатка.
;
        ADDL3   R1,R0,R4        ;Вычислить адрес нового отрезка
        MOVL    (R1)+,(R4)+     ;Скопировать связь вперед из
                                ;старого отрезка в новом отрезке
        SUBL3   R0,(R1),(R4)    ;Запомнить размер нового отрезка
                                ;во втором слове удвоенной длины
        MOVAL   -(R4),-(R1)     ;Переслать адрес нового отрезка
                                ;в вершину старого блока
20$:    MOVL    (R1),(R3)       ;Связать новый блок с предыдущим
        MOVL    R1,@RETADR(AP)  ;Вернуть адрес выделенного блока
        MOVL    #1,R1           ;Пометить успешное завершение
        RET                     ;Возврат

30$:    CLRL    R1              ;Пометить
                                ;неудачное завершение
        RET                     ;Возврат в вызывающую программу
```

Рис. 4.11. Процедура выделения памяти

Операции добавления и удаления элементов очереди выполняются сложнее, чем в случае однократно связного списка, потому что должны быть модифицированы три элемента с указателями: добавляемый или удаляемый, его предшественник и его преемник.

Действия при добавлении элемента состоят в запоминании следующих адресов:

преемника в новом элементе в поле связи вперед;

предшественника в новом элементе в поле связи назад;

Рис. 4.12. Блок-схема выделения динамической памяти

нового элемента в предшественнике в поле связи вперед;
нового элемента в преемнике в поле связи назад.

Действия при удалении элементов состоят в запоминании следующих связей:

вперед из удаляемого элемента в поле связи вперед его предшественника;

назад из удаляемого элемента в поле связи назад его преемника.

Необходимо заметить, что элемент может быть удален из двукратно связного списка при указании только адреса этого элемента, в то время как при удалении из однократно связного списка — адреса предшествующего элемента.

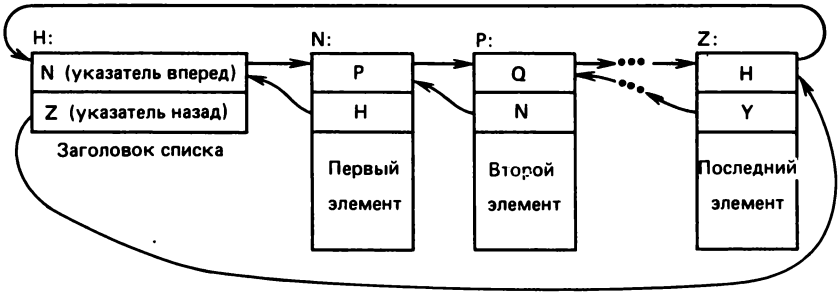


Рис. 4.13. Двукратно связный список

Операции с очередями используются часто, поэтому архитектура СМ1700 предусматривает специальные инструкции INSQUE и REMQUE для добавления (удаления) элементов в (из) двукратно связные списки. Они предназначены главным образом для операционных систем, динамические структуры которых содержатся в виде очередей. Форматы этих инструкций таковы: INSQUE — элемент, предшественник; REMQUE — элемент, преемник.

Инструкция INSQUE добавляет указанный элемент в очередь следом за предшественником. Оба операнда (элемент и предшественник) адресуются из заголовков очереди. Инструкция REMQUE удаляет элемент из очереди и помещает его адрес в указанный преемник.

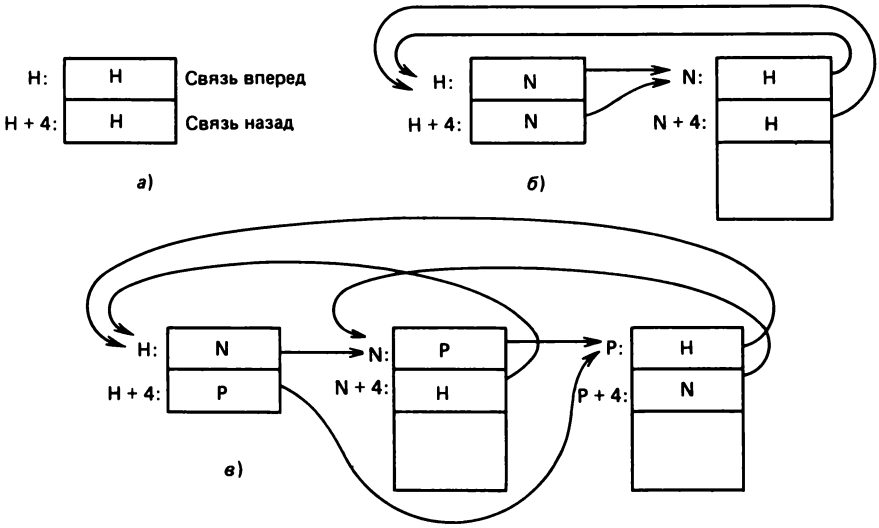


Рис. 4.14. Заголовок пустой очереди (*а*) и очереди с одним (*б*) и двумя (*в*) элементами

Рисунок 4.14 иллюстрирует сказанное. Чтобы вставить элемент N в начало очереди следом за заголовком, можно использовать инструкцию INSQUE N,H. Затем, для того чтобы добавить элемент P следом за последним элементом, т. е. в конец очереди, необходимо использовать инструкцию INSQUE P,@H+4. Спецификация @H+4 использует указатель конца в заголовке очереди для ссылки на адрес последнего элемента как на предшественник при операции добавления к очереди. Поскольку последним элементом был N, можно было бы с тем же успехом использовать инструкцию INSQUE P,N (вставить элемент P следом за элементом N). Однако рекомендуется делать добавление в конец очереди, используя указатель в заголовке очереди.

При удалении элементов инструкция REMQUE @H,TEMP удаляет первый элемент очереди, загружая его адрес в двойное слово TEMP. Инструкция REMQUE @H+4,TEMP удаляет последний элемент, загружая его адрес в ячейку TEMP.

Инструкция INSQUE устанавливает бит Z кодов условия, если в очередь был вставлен первый элемент. Инструкция REMQUE устанавливает бит Z, если из очереди был удален последний элемент, и бит V, если не оказалось элемента, предназначенного для удаления. Таким образом можно легко опросить специфические условия очереди.

Важным свойством инструкций для очередей является то, что они не могут быть прерваны. Очереди часто используются для коммуникации или синхронизации сразу нескольких процессов. Если бы для операций вставки и удаления элементов требовалось несколько шагов программы, то не исключена ситуация, когда процесс может быть прерван до завершения операции (при этом указатель очереди остается в неопределенном состоянии). Поскольку обработка очереди выполняется с помощью одиночных непрерываемых инструкций, то прерывания не влияют на взаимодействие процессов, находящихся в данной очереди.

Программа, которая приведена в приложении 4, является примером использования очередей двумя взаимодействующими процессами. Первый процесс собирает экспериментальные данные и добавляет их в буфер. Поскольку данные поступают в буфер быстро, процесс не может допустить, чтобы при записи буфера на диск терялось время. Поэтому он ставит буфер в очередь ко второму процессу, который записывает данные на диск. Существует очередь свободных буферов. Когда первый процесс заполняет буфер, он берет новый буфер из очереди свободных буферов. Когда второй процесс освобождает буфер, он помещает его в очередь свободных буферов для использования его процессом сборщиком информации.

Второй процесс удаляет буферы из очереди и записывает их на диск. Когда процесс находит очередь пустой, он приостанав-

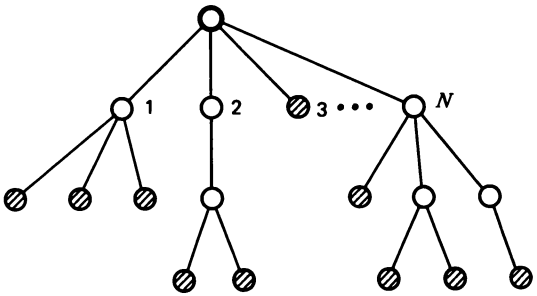


Рис. 4.15. Простое дерево

ливается или, как говорят, «спит». Первый процесс «будит» его, когда новый буфер помещается в очередь. Эти процессы должны разделять память, которая содержит заголовки очередей и буферы данных.

Программа содержит три процедуры: INIT — инициализации, которая устанавливает в начальное состояние очереди, заголовки очередей и соответствующие переменные; COLDAT — сбора данных; OUTPUT — записи из второго процесса.

Деревья. Дерево — это простая связная структура (рис. 4.15), в которой существует один специальный элемент, называемый корнем, функции которого аналогичны заголовку очереди в связном списке. Корень указывает на N элементов, каждый из которых сам может являться деревом.

Отношения между элементами (узлами) дерева часто описывают, используя терминологию генеалогического дерева: родитель, потомок, предок и т. п. Концевые узлы называются листьями. Дерево на рис. 4.15 имеет корень наверху. Отношения «верх», «низ», «лево» и «право» используются для последовательного описания элементов дерева.

Деревья часто полезны для запоминания в памяти представлений альтернатив, особенно в «играх», где каждый путь представляет возможное перемещение или действие. Например, вслед за каждым ходом при игре в «крестики — нолики» некое поддерево описывает оставшиеся возможные ходы и позволяет принять решение, какой ход приводит к победе или поражению. Конечно, для более сложных игр существенно увеличивается перебор всех вариантов в полном дереве.

Наиболее принятой разновидностью дерева является двоичное, в котором каждый узел может иметь от нуля до двух потомков. Двоичное дерево обычно упорядочено. Например, дерево может быть построено так, что для значения К, запомненного в каком-либо узле, все узлы-потомки со значением меньше К являются левыми в дереве, а все узлы со значением больше К — правыми. Двоичные деревья часто используются для построения таблиц

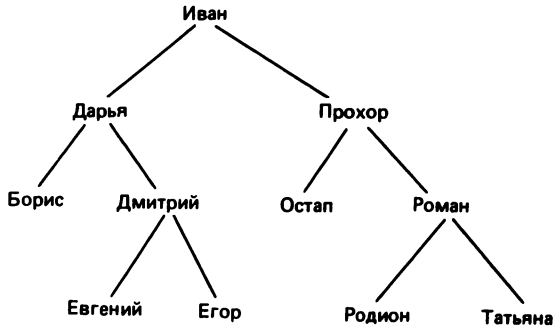


Рис. 4.16. Алфавитное двоичное дерево

имен, поскольку такое упорядочивание облегчает поиск и распечатку в алфавитном порядке.

Например, необходимо построить дерево для запоминания следующих имен: Иван, Прохор, Дарья, Роман, Дмитрий, Борис, Егор, Татьяна, Остап, Евгений, Родион. Если начать с имени Иван как с корня дерева и добавлять каждое последующее имя так, чтобы по алфавиту оно располагалось справа или слева относительно первого имени, то дерево будет иметь вид, показанный на рис. 4.16.

Процедура TREE (рис. 4.17) добавляет элемент в двоичное дерево. Каждый элемент является блоком из пяти слов удвоенной длины, в котором первые два слова используются для указания на двух потомков (левого и правого), а последние три слова содержат 12-байтовую строку имени. Процедура просматривает дерево вниз до тех пор, пока не находит конечный узел, после которого может быть добавлен новый элемент. Нулевой указатель сигнализирует, что потомков нет.

Глава 5. АППАРАТНЫЕ СРЕДСТВА СМ1700, ПОДДЕРЖИВАЮЩИЕ ОПЕРАЦИОННУЮ СИСТЕМУ

5.1. РАСПРЕДЕЛЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ РЕСУРСОВ

Операционная система должна обеспечить мультипрограммный режим выполнения программ, которые разделяют общие вычислительные ресурсы ЭВМ: процессор, оперативную память и устройства ввода-вывода. Таким образом, она должна создавать

```
;       TREE - Эта процедура добавляет блок из 5 длинных слов,
;       содержащий строку имени в коде СМ1700, в двоичное дерево.
;       Двойное слово ROOT содержит адрес корневого узла.
;
;  Входные параметры:
;       BLOCK(AP) - содержит адрес добавляемого узла из пяти
;                   двойных слов,имеющего структуру:
;                4 байт:        связь влево
;                4 байт:        связь вправо
;               12 байт:        строка имени
;
;  Возвращаемые значения:
;       R0=1 -- узел добавлен успешно
;       R0=0 -- имя уже существует в дереве
;
        BLOCK=4                 ;Смещение до аргумента
        LEFT=0                  ;Смещение до связи влево
        RIGHT=4                 ;Смещение до связи вправо
        NAME=8                  ;Смещение до строки "имя"
TREE:
        .WORD   ^M<R2>          ;Сохранить R2
        PUSHL   #0              ;Загрузить код ошибки,
                                ;предполагая дублирование
        MOVL    BLOCK(AP),R0    ;Адрес входного блока
        MOVAL   ROOT,R2         ;Получить адрес корня
        BRB     20$             ;Проверить существование корня
;
; Проверка на то, что новый элемент <,> или = (меньше, больше,
; или равен) текущему блоку. Если <или>, то продолжить
; сканирование дерева или вставить новый элемент, если нет
; преемника текущего узла. Если =, то возврат с индикацией ошибки.
; Когда дерево просканировано полностью, R0 содержит адрес нового
; элемента, который должен быть вставлен, а R1 содержит адрес узла
; в дереве, который был исследован последним.
;
10$:    MOVAL   LEFT(R1),R2     ;Указатель к < (левой) стороне
        CMPC3   #12,NAME(R0),NAME(R1) ;Сравнить "имя" с именем узла
        BEQL    40$             ;Переход, если совпадают
        BLSS    20$             ;Продолжить, если < текущего узла
        MOVAL   RIGHT(R1),R2    ;Получить указатель к > стороне
20$:    TSTL    (R2)            ;Есть ли потомок этого узла?
        BEQL    30$             ;Если нет, вставить новый элемент
        MOVL    (R2),R1         ;Иначе получить адрес потомка и
        BRB     10$             ;Продолжить сканирование дерева
;
;  Найти место для вставки нового узла. R2 содержит адрес указателя
;  в новом родителе.
;
30$:    CLRQ    (R0)            ;Обнулить оба указателя в новом узле
        MOVL    R0,(R2)         ;Связать новый узел с родительским узлом
        PUSHL   #1              ;Загрузить в стек признак успеха
40$:    POPL    R0              ;Извлечь из стека признак завершения
        RET                     ;Возврат
```

Рис. 4.17. Процедура добавления элемента в двоичное дерево

для программ логическую среду, которая и обеспечивает простой способ использования вычислительных ресурсов пользователями. На самом деле такая простота достигается невидимой для пользователя сложной структурой операционной системы. Эта сложность объясняется многообразием и противоречивостью

функций, выполняемых операционной системой: во-первых, организацией ввода-вывода на физическом уровне с обеспечением контроля всех возникающих ошибочных ситуаций; во-вторых, предоставлением общих вычислительных ресурсов каждой программе с обеспечением защиты программ от несанкционированного доступа друг к другу; в-третьих, распределением вычислительных ресурсов для каждой программы с обеспечением в целом высокой загрузки ЭВМ.

Распределение времени процессора, как и других вычислительных ресурсов, между программами осуществляется операционной системой динамически. При этом время процессора может разделяться между программами квантами времени, выделяемыми каждой программе, или на основе принципа приоритетного выполнения программ. Первый способ характерен для режима разделения времени, а второй — для режима реального времени. В обоих случаях при завершении текущей программы или переходе ее в режим ожидания завершения операции ввода-вывода для выполнения на процессоре выбирается другая программа. Конечно, при мультипрограммном режиме время, выделяемое каждой программе, уменьшается, т. е. каждая программа выполняется с меньшей скоростью, чем в однопрограммном режиме. Однако для пользователей, работающих в интерактивном режиме, за терминалами это замедление не ощущается, так как операции ввода-вывода совмещаются с выполнением программ других пользователей.

Распределение оперативной памяти между несколькими программами является более сложной процедурой в операционной системе, чем распределение времени процессора. Это объясняется тем, что операционная система и аппаратные средства кроме разделения памяти между программами должны создать логическую среду, в которой выполняются эти программы. Такая логическая среда обеспечивает представление каждой программы в непрерывном логическом адресном пространстве, начинающемся с нулевого адреса. Генерируемые программой адреса, относящиеся к непрерывному логическому адресному пространству, называются виртуальными, так как они не относятся к физическим адресам. При выполнении программы каждый виртуальный адрес преобразуется (транслируется) в физический. Это преобразование осуществляется аппаратно с помощью диспетчера памяти.

Размер виртуального адресного пространства определяется разрядностью ячейки оперативной памяти. В СМ1700 разрядность ячейки 32 разряда обеспечивает максимальный адрес 2^{32}, что и определяет размер виртуального адресного пространства объемом в 4 млрд. байт. Объем физической памяти в СМ1700 значительно меньше (не более 5 Мбайт), таким образом программе предоставляется возможность обращения за пределы физически доступной памяти. Это достигается использованием старших виртуальных адресов для обращения к адресному пространству на магнитных дисках. В этом случае обращение к таким виртуальным адресам из программы вызывает загрузку с диска в оперативную память тех частей программы, которые находятся на диске.

Таким образом операционная система создает для каждой программы логическую среду и обеспечивает управление памятью, разделяемой между

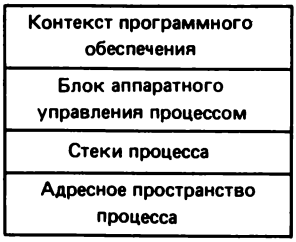


Рис. 5.1. Структура процесса

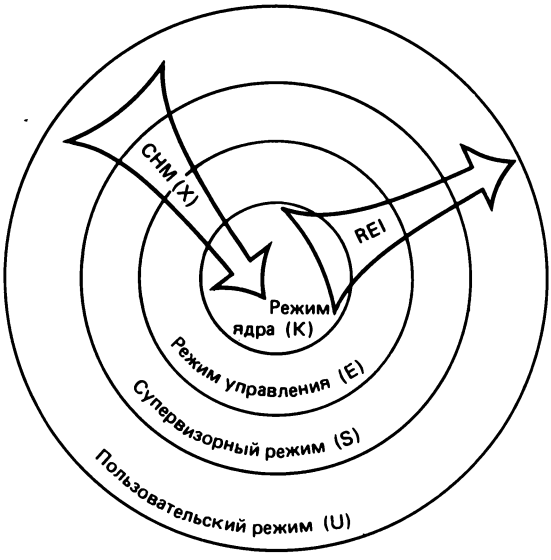


Рис. 5.2. Режимы доступа к процессору

многими программами. При этом каждая программа выполняется так, как будто она единственная и загружена в непрерывную физическую память.

Процессы. Единицей планируемой работы в операционной системе СМ1700 является процесс. Каждый процесс определяется совокупностью контекстов аппаратных средств и программного обеспечения и описанием адресного пространства. Процесс является средой, в которой выполняется образ (т. е. скомпонованная и готовая к выполнению программа). Процесс можно сопоставить с виртуальной машиной, которая определяет адресное пространство и логические ресурсы, выделенные пользователю. При этом виртуальная машина отождествляется с концептуальной средой, с которой взаимодействует программа, совпадающей или не совпадающей с физическими ресурсами. Информация о каждом процессе хранится в нескольких структурах данных, расположенных в различных областях адресного пространства, выделенного процессу.

На рис. 5.1 показана в общем виде структура процесса. Контекст программного обеспечения содержит управляющую информацию о процессе, используемую операционной системой для планирования и управления процессом при его выполнении: приоритет процесса; привилегии и ресурсы процесса; идентификатор процесса и др.

Блок аппаратного управления содержит контекст аппаратных средств процесса: значения регистров процессора; указатель стеков; значения регистров процессора, используемых для управления памятью; слово состояния процессора; значения регистров, используемых для обработки асинхронных системных прерываний.

Рассмотрим, как архитектура СМ1700 поддерживает операционную систему при разделении процессов и защиту их друг от друга.

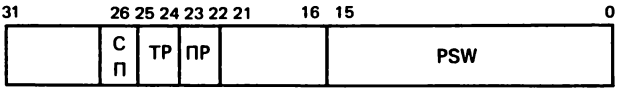
Режимы доступа к процессору. Процессор СМ1700 выполняет команды в одном из четырех режимов: ядра; управления; супервизора; пользователя. В соответствии с этим операционная система обеспечивает обслуживание процессов в одном из четырех режимов доступа к процессору, а процессор — механизм защиты программ, основанный на четырех режимах доступа к процессору. Каждый режим процесса определяет его привилегии в части доступа к памяти и возможность использования различных команд. Например, команда HALT (останов процессора) может быть выполнена только в режиме ядра (наиболее привилегированном).

Различные режимы доступа к процессору обеспечивают взаимную защиту компонентов операционной системы, выполняемых на различных уровнях. Структура операционной системы (рис. 5.2) характеризуется наличием нескольких функциональных уровней, расположенных один над другим. Каждый уровень обеспечивает выполнение функций, используемых следующим верхним уровнем. Разделение операционной системы на уровни упрощает ее структуру. Если четко определены интерфейсы между уровнями, то обеспечивается независимость разработки каждого уровня операционной системы.

Таким образом в СМ1700 обеспечивается выделение в операционной системе четырех уровней, защищаемых аппаратно от доступа менее привилегированных уровней к более привилегированным. Программные компоненты операционной системы, выполняемые в одном из режимов, имеют защиту от непосредственного доступа программ, выполняемых в менее привилегированном режиме. Подобным образом ограничивается доступ к некоторым структурам данных операционной системы (он разрешается только в наиболее привилегированных режимах). Основные функции операционной системы распределяются в соответствии с доступом следующим образом: пользовательские и системные обслуживающие программы выполняются в пользовательском режиме; интерпретация команд оператора — в режиме супервизора; управление файлами — в режиме управления; функция планирования процессов, управление ресурсами и вводом-выводом — в режиме ядра.

Стеки. Каждый выполняемый процесс имеет четыре стека, используемые в зависимости от текущего режима доступа к процессору. Каждый стек защищается от доступа к нему из менее привилегированного режима. Введение отдельных стеков для каждого режима позволяет сохранять информацию в текущем стеке без его защиты при изменении режима процесса на менее привилегированный.

Рис. 5.3. Формат длинного слова состояния процессора



Значения указателей стека для четырех режимов доступа текущего процесса хранятся в привилегированных регистрах процессора KSP, ESP, SSP, USP. При переходе процесса из одного режима в другой регистр указателя стека (SP) автоматически переключается на указатель стека нового режима процессора. Кроме четырех стеков, используемых процессом в одном из четырех режимов доступа к процессору, существует стек прерываний, используемый системой для обслуживания прерываний. Указатель стека прерываний является общим для всех режимов выполнения процесса. Регистр указателя стека SP переключается на стек прерываний автоматически, когда происходит прерывание.

Переключение режимов процессора. Часть информации о состоянии процессора хранится в аппаратном регистре, называемом длинным словом состояния процессора (PSL). Формат регистра PSL представлен на рис. 5.3.

Старшие 16 разрядов PSL защищаются и не могут модифицироваться программой пользователя, т. е. отражают привилегированное состояние процессора. Младшие 16 разрядов являются обычным словом состояния процесса (PSW). Двухразрядные поля ПР и ТР в старших разрядах PSL отражают предыдущий и текущий режимы процессора в соответствии со следующими значениями этих полей:

00 — режим ядра;
01 — режим управления;
10 — режим супервизора;
11 — пользовательский режим.

Одноразрядное поле СП в PSL указывает, какой стек используется при обработке прерываний. Если СП = 1, процессор использует стек прерываний, при этом в поле текущего режима ТР должно быть установлено значение 00, т. е. процессор должен находиться в режиме ядра. Если СП = 0, процессор для обработки прерываний использует стек, определяемый текущим режимом процессора.

Многоуровневая структура операционной системы, как уже указывалось, обеспечивает выполнение различных функций системы в разных режимах процессора. Если программа с некоторой функцией должна выполняться в более привилегированном режиме, требуется изменение режима процессора. Например, запрос из пользовательской программы на ввод-вывод непосредственно реализуется в режиме ядра. Переключение режимов процессора осуществляется с помощью специальных команд изменения режимов доступа CHMK, CHME, CHMS, CHMU, обеспечивающих соответственно переходы в режимы ядра, управления, супервизора и пользовательский (из менее привилегированного режима в более привилегированный). Эти команды имеют один

операнд, являющийся кодом для определения привилегированной функции или операции, которую требуется выполнить. При выполнении команд изменения режимов осуществляется переключение на стек указанного режима. Содержимое PSL и PC для предыдущего режима процесса сохраняется в текущем (новом) стеке. Код предыдущего режима процессора сохраняется в поле предыдущего режима, а код нового режима устанавливается в поле текущего режима PSL. Таким образом в более привилегированном (новом) режиме можно определить предыдущий режим. Программа, получившая управление в новом режиме доступа к процессору, анализирует передаваемый в команде код операнда и передает управление соответствующему компоненту операционной системы для выполнения запрошенной функции.

Например, команда CHMK 15 требует выполнения в режиме ядра процедуры, определяемой кодом 15. Обычно в пользовательской программе команды изменения режима не используются явно, а для выполнения требуемой системной функции осуществляется вызов процедуры. Вызываемая процедура сама выполняет команды изменения режима. Необходимо отметить, что системная программа, реализующая требуемую функцию, выполняется при этом в контексте процесса пользователя. Хотя системные программы используются в более привилегированном режиме (большинство в режиме ядра), они имеют доступ ко всему адресному пространству пользовательского процесса. В этом смысле системные программы не отличаются от обычных подпрограмм. Команды изменения режима доступа к процессору можно рассматривать как обращения к этим программам, выполняемым в более привилегированном режиме. Так как все команды изменения режимов контролируются специальной системной программой (диспетчером), возможность некорректного использования команд изменения режимов исключается.

При завершении выполнения системных программ единственным способом перехода к менее привилегированному режиму (возврату управления к пользовательской программе) является выполнение команды выхода из прерываний REI (рис. 5.2, где X в команде CHM(X) может принимать значения U, S, E, K). При выполнении команды REI в вершине текущего стека должны находиться PSL и PC для нового (менее привилегированного) режима. С помощью анализа поля текущего режима в PSL, находящегося в стеке, проверяется, является ли новый режим менее привилегированным по сравнению с текущим, при этом не допускается переход по команде REI к более привилегированному режиму. Содержимое PSL и PC, извлекаемых из стека по команде REI, определяет после выполнения команды новое состояние процессора и соответственно переключение на указатель стека нового режима, а также адрес выполнения программы в новом режиме.

Команды изменения режимов доступа к процессору и возврата из прерывания являются механизмом управления привилегиями процесса. Следовательно, операционную систему можно рассматривать как набор системных программ, выполняемых в контексте процесса пользователя в более привилегированных режимах, чем пользовательский. Если пользователь не имеет непосредственного доступа к привилегированным ресурсам, то системные программы обеспечивают выполнение для пользователя различных привилегированных функций.

Контроль доступа процесса к памяти. Команды изменения режима доступа обеспечивают обращение к системным программам, выполняемым в более привилегированном режиме. Входные параметры для такого обращения передаются системным программам чаще всего в стеке, выходные — системными программами в буфере программы пользователя, адрес которого определяется в качестве одного из входных параметров. При отсутствии специальных средств в некоторых случаях при неправильном задании адреса буфера может возникнуть ситуация, приводящая к аварийному завершению работы операционной системы. Например, если в программе пользователя ошибочно будет указан адрес буфера в области адресного пространства операционной системы, то системная программа, выполняя требуемую функцию, сможет модифицировать содержимое области памяти системы, так как она выполняется чаще всего в режиме ядра.

Для исключения подобных ситуаций в архитектуре СМ1700 предусмотрены две команды для проверки возможности обращения вызывающей программы к указанным адресам памяти. Команда PROBER обеспечивает проверку допустимости чтения, а команда PROBEW — проверку допустимости записи. С помощью этих команд системная программа, выполняемая в более привилегированном режиме, осуществляет контроль возможности обращения вызывающей программы к адресам памяти, указанным как входные параметры.

Переключение контекста процесса. В операционной системе СМ1700, как в любой другой мультипрограммной системе, часто возникает необходимость передиспетчеризации процессов. Например, при переходе текущего процесса в режим ожидания ввода-вывода или истечении выделенного текущему процессу кванта времени операционная система должна обеспечить сохранение состояния текущего процесса и передать управление другому. Процедура, связанная с передачей управления, называется переключением контекста процессов. Время, необходимое для выполнения этой процедуры, может заметно влиять на производительность системы. Это связано с необходимостью сохранения большого объема аппаратного контекста текущего процесса и загрузки такого же аппаратного контекста для нового процесса.

```
;Пример переключения контекста процесса
;Вход в программу происходит по прерыванию
 START:
        SVPCTX          ;Сохранить состояние
                        ;текущего процесса в
                        ;текущем PCB
;Выбрать из очереди новый процесс для
;обслуживания и загрузить в R1 физический
;адрес его блока управления процессом
        MTPR R1, #16.   ;Загрузить в PCBB
                        ;адрес PCB нового процесса
        LDPCTX          ;Загрузить аппаратный
                        ;контекст нового процесса,
                        ;а также поместить
                        ;в стек прерываний PC
                        ;и PSL нового процесса
        REI             ;Запустить на выполнение
                        ;новый процесс
```

Рис. 5.4. Пример переключения контекста процесса

KSP
ESP
SSP
USP
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12(AP)
R13(FP)
PC
PSL
P0BR
P0LR
P1BR
P1LR

Рис. 5.5. Блок управления процессом

В архитектуре СМ1700 предусмотрена аппаратная реализация процедуры переключения контекста процессов.

Аппаратный контекст процесса в системе определяется блоком управления процессом (PCB) (рис. 5.4), представляющим структуру данных, в которой хранятся образы 14 универсальных регистров (R0 — R13), четырех указателей стека для каждого режима доступа к процессору (KSP, ESP, SSP, USP), длинного слова состояния процессора (PSL), счетчика команд (PC), регистров, используемых для управления памятью процесса (P0BR, P0LR, P1BR, P1LR), и другие управляющие поля. При переключении контекста процессов содержимое аппаратных регистров для текущего процесса сохраняется в PCB этого процесса, а аппаратные регистры загружаются значениями (образами) этих регистров из PCB нового процесса. Адрес PCB текущего процесса всегда хранится во внутреннем привилегированном регистре СМ1700, называемом регистром базы блока управления процессом (PCBB). Процедура сохранения состояния текущего процесса осуществляется с помощью команды SVPCTX, которая реализует

сохранение всех регистров в блоке PCB, адрес которого содержится в регистре PCBB. Затем в регистр PCBB загружается адрес PCB для нового процесса и с помощью команды LDPCTX обеспечивается загрузка регистров из PCB нового процесса. Так как после выполнения команды SVPCTX операционная система работает в режиме ядра со стеком прерываний (из-за отсутствия в этот момент выполняемых процессов), то запуск на выполнение нового процесса после загрузки его аппаратного контекста командой LDPCTX осуществляется по команде REI. Поэтому команда LDPCTX дополнительно заносит в стек прерывания значения регистров PC и PSL для нового процесса.

На рис. 5.5 приведена последовательность команд для процедуры переключения контекста. Команда MTPR предназначается для загрузки внутренних регистров процессора, в примере — для загрузки регистра PCBB адресом PCB для нового процесса.

Поскольку каждый процесс в системе использует свои стеки и указатели стеков для всех возможных режимов доступа к процессору, переключение контекста процессов может выполняться даже тогда, когда в контексте процесса в режиме ядра выполняется некоторая системная программа. Это является отличительной особенностью архитектуры СМ1700 по сравнению с архитектурами других ЭВМ, в которых операционная система всегда выполняется в особом режиме (без возможности ее прерывания).

5.2. УПРАВЛЕНИЕ ПАМЯТЬЮ

Адресное пространство является частью контекста каждого процесса. Управление доступом к памяти процесса осуществляется системными программами и поддерживающей эти программы аппаратурой (диспетчером памяти). Управление памятью в операционной системе должно обеспечить каждому пользователю возможность использования непрерывного пространства адресов памяти, начиная с нулевого адреса, а также распределения физической памяти между процессами. Кроме этого оно должно контролировать возможность обращения пользователей к памяти с различными режимами доступа (чтение или запись). С целью эффективного использования памяти операционная система должна обеспечивать одновременное нахождение в памяти нескольких процессов.

Организация памяти. Виртуальное адресное пространство в СМ1700, определяемое 32-разрядным адресом, представляется для пользователя непрерывным пространством объемом 2^{32} байт. Виртуальное адресное пространство является настолько большим, что, естественно, не может быть отображено в реальной физической (оперативной) памяти. Поэтому диспетчер памяти отображает в физической памяти только часть виртуального адресного пространства процесса.

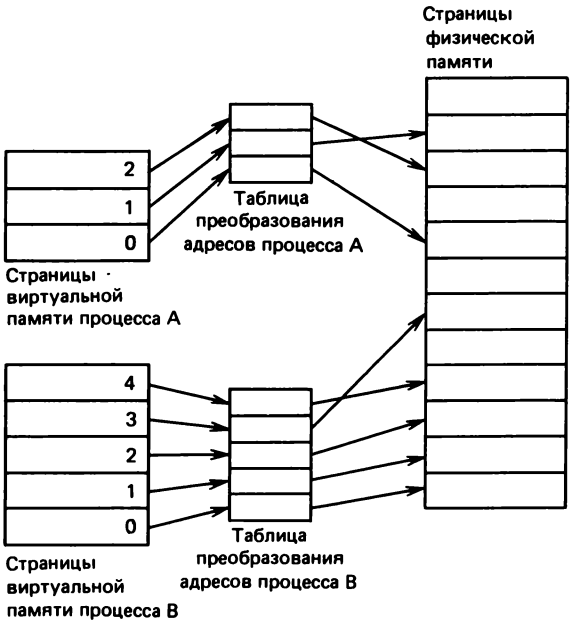
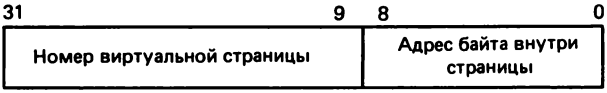


Рис. 5.6. Общая схема преобразования виртуальных адресов

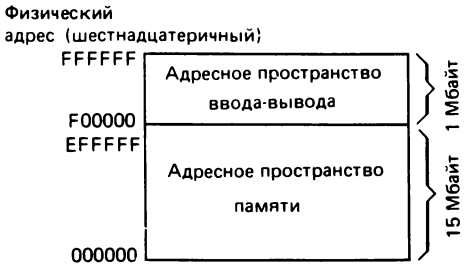
Виртуальное адресное пространство разделяется на две функциональные области: размещения процессов и операционной системы. Область памяти системы является общей для всех процессов, доступ к ней из программы пользователя или других обслуживающих программ осуществляется через вызовы системных процедур. Каждый процесс имеет свое собственное адресное пространство, однако несколько процессов могут обращаться к общему пространству памяти операционной системы. Таким образом обеспечивается разделение памяти между процессами.

Виртуальное адресное пространство разделяется на части по 512 байт, называемые страницами. Программа строится в виде последовательного набора страниц, пронумерованных от нуля до некоторого значения. Так как размер страницы 512 байт, для адресации байта внутри страницы используется 9 бит 32-разрядного виртуального адреса. Формат виртуального адреса представлен ниже:



Старшие разряды (с 9-го по 31-й) виртуального адреса определяют номер виртуальной страницы. Адресация байтов

Рис. 5.7. Распределение физического адресного пространства



в странице и нумерация виртуальных страниц начинается с 0. Например, десятичный виртуальный адрес 521 указывает байт номер 9 в виртуальной странице номер 1.

Отображение виртуального адресного пространства в физической памяти осуществляется диспетчером памяти, который кроме преобразования виртуальных адресов в физические обеспечивает распределение памяти между процессами и управление доступом к памяти, т. е. ее защиту. Физическая память в СМ1700 также разделяется на страницы по 512 байт. Общая структура отображения виртуальных адресов процессов в физической памяти представлена на рис. 5.6. Адресация физической памяти выполняется с помощью 24-разрядного физического адреса. Младшие 9 разрядов физического адреса указывают номер байта в странице, а старшие 15 разрядов — номер страницы в физической памяти.

Физическое адресное пространство в СМ1700 (рис. 5.7) разделено на две области: памяти и ввода-вывода. Физический 24-разрядный адрес позволяет адресовать физическое адресное пространство размером 16 Мбайт, однако конструктивно объем оперативной памяти ограничен 5 Мбайт. Виртуальное адресное пространство в СМ1700 (рис. 5.8) разделяется на две половины. Младшая половина виртуальной памяти используется для адресации процессов (пространство процессов), старшая половина — системой (системное пространство). Пространство процессов,

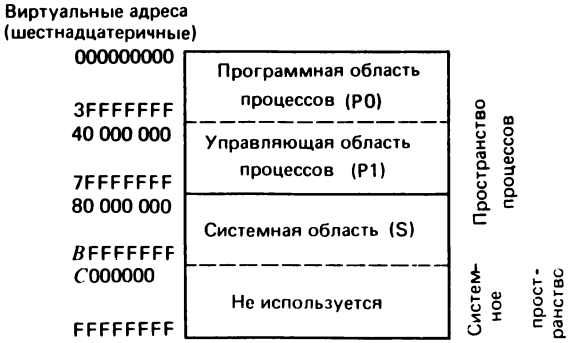


Рис. 5.8. Распределение виртуального адресного пространства

в свою очередь, делится на две равные части: программную (P0) и управляющую (P1) области процессов.

Старшая половина системного пространства не используется. Два старших разряда виртуального адреса определяют область, к которой принадлежит адрес: 00 относятся к P0-области; 01 к P1-области; 10 — к области системы; 11 — не используется.

Преобразование виртуального адреса в физический. Отображение виртуального адресного пространства в физической памяти осуществляется с помощью диспетчера памяти. Виртуальная страничная организация памяти позволяет хранить в физической памяти только часть страниц процесса, а другую часть страниц, которые в данный момент не используются,— на диске. Страницы с диска будут загружаться в физическую память при обращении к ним, т. е. по мере необходимости, замещая те страницы, которые не используются. Такая процедура загрузки страниц с диска называется обменом страниц. Страницы физической памяти, выделяемые для процесса, могут быть несмежными, т. е. находиться в разных местах физической памяти. Однако при выполнении программы диспетчер памяти должен каждый виртуальный адрес, генерируемый процессором, преобразовывать (транслировать) в соответствующий физический адрес. Для трансляции виртуальных адресов в физические используется структура данных, называемая таблицей страниц, которая содержит информацию, необходимую для трансляции адресов. Таблица страниц представляет набор записей, по одной на каждую страницу виртуальной памяти. Каждая запись имеет размер двойного слова и указывает:

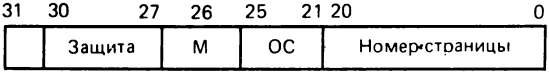
а) режим доступа процессора (чтение или запись) к соответствующей странице физической памяти;

б) расположение страницы — в физической памяти или на диске;

в) номер страницы в физической памяти, если она находится в памяти.

Если данная страница не находится в памяти, запись в таблице страниц определяет адрес образа этой страницы на диске.

Формат записи в таблице страниц представлен ниже:



Разряд 31 в записи называется разрядом достоверности. Единичное значение этого разряда указывает, что соответствующая страница находится в физической памяти. Разряды 0 ... 20 записи содержат номер страницы в физической памяти. Нулевое значение разряда достоверности указывает, что требуемая страница находится на диске. Обращение по виртуальному адресу, отображаемому в страницу, находящуюся на диске (т. е. отсут-

ствующую в физической памяти), приводит к передаче управления операционной системе для загрузки этой страницы с диска в физическую память. Разряды 0—20 в этом случае определяют адрес страницы на диске. Разряд 26 определяет, выполнялась ли модификация соответствующей страницы. Этот разряд устанавливается аппаратно при выполнении первой операции записи (модификации) в данную страницу. Операционная система использует этот разряд модификации для определения необходимости выполнения выгрузки страницы на диск. Если страница не модифицировалась, то разряд 26 остается нулевым и выгрузки соответствующей страницы на диск не требуется, так как на диске в образе процесса уже имеется образ этой страницы. Разряды 27 ... 30 записи в таблице страниц содержат код защиты, который определяет доступность страницы в различных режимах процессора. Возможны три типа доступа к странице: чтение — запись (RW), чтение (R) и страница недоступна для обращения к ней. В табл. 5.1 приведены возможные коды защиты для различных режимов доступа процессора; при нарушении привилегий доступа аппаратно вырабатывается ошибка нарушения доступа. Разряды 21 ... 25 в записи используются операционной системой.

Диспетчер памяти использует таблицу страниц при каждом обращении к памяти. Номер виртуальной страницы в виртуальном адресе определяет конкретную запись в таблице страниц. При единичном значении бита достоверности и соответствующем коде защиты адрес страницы в физической памяти определяется

Таблица 5.1. **Коды защиты для различных режимов доступа процессора**

Коды защиты (разряды 27 30)	Режимы доступа процессора			
	K	E	S	U
0000	—	—	—	—
0001	Не предсказуем			
0010	RW	—	—	—
0011	R	—	—	—
0100	RW	RW	RW	RW
0101	RW	RW	—	—
0110	RW	R	—	—
0111	R	R	—	—
1000	RW	RW	RW	—
1001	RW	RW	R	—
1010	RW	R	R	—
1011	R	R	R	—
1100	RW	RW	RW	R
1101	RW	RW	R	R
1110	RW	R	R	R
1111	R	R	R	R

Примечание. « » отсутствие доступа к странице.

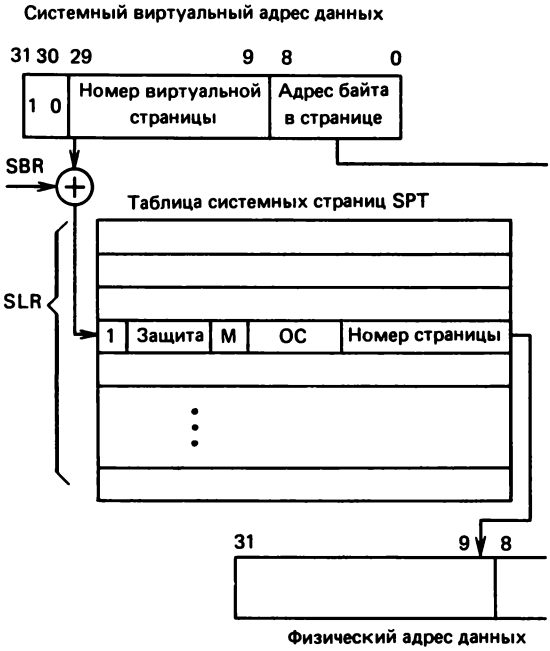


Рис. 5.9. Преобразование системного виртуального адреса

конкатенацией младших 21 разрядов записи в таблице страниц и младших 9 разрядов виртуального адреса.

Трансляция адреса в пространстве системы. Виртуальное адресное пространство системы определяется двоичным значением 10 в разрядах 30, 31 виртуального адреса. Для пространства системы существует своя таблица страниц, называемая таблицей системных страниц (SPT), которая используется для трансляции всех обращений в пространство системы. Все обращения процессов к некоторому виртуальному адресу в пространстве системы будут отображаться на один и тот же адрес в физической памяти. Так как все процессы разделяют операционную систему, которая выполняется в системном пространстве, то система будет отображаться в одинаковых виртуальных адресах для всех процессов.

Таблица системных страниц (SPT) описывается двумя аппаратными регистрами. Базовый регистр системы (SBR) содержит начальный физический адрес таблицы системных страниц, которая должна быть непрерывной. Обращение диспетчера памяти к SPT всегда осуществляется по физическим адресам, так как процедура трансляции не может быть выполнена при отсутствии SPT. Регистр длины системы (SLR) определяет размер таблицы системных страниц в двойных словах. Процедура трансляции системного виртуального адреса в физический представлена на рис. 5.9.

Физический адрес записи в SPT вычисляется с использованием содержимого SBR и номера виртуальной страницы в системном виртуальном адресе. Физический

адрес данных определяется конкатенацией разрядов 0...20 записи в SPT и разрядами 0...8 виртуального адреса.

Пространство процесса. Виртуальное адресное пространство процесса разделяется на две равные области. Нулевое значение 30-го разряда в виртуальном адресе определяет принадлежность адреса к P0-области, а единичное значение этого разряда — к P1-области. Первая область используется обычно для адресации программ, начинается с виртуального нулевого адреса и расширяется в направлении старших адресов, вторая — для размещения управляющей информации о процессе, используемой операционной системой, а также для размещения стеков процесса. Эта вторая область начинается с самого старшего адреса ($2^{31}-1$) и расширяется в направлении младших адресов. В отличие от пространства системы, которое совместно используется (разделяется) всеми процессами, пространство процесса полностью принадлежит каждому процессу в системе. Это означает, что пространство каждого процесса описывается своими таблицами страниц P0 (P0PT) и P1 (P1PT), существующими для обеих областей. В связи с этим обращения разных процессов к одному и тому же виртуальному адресу в пространстве процессов отображаются на различные физические адреса. Для преобразования виртуальных адресов в пространстве процесса используются регистры базы P0BR и P1BR и регистры длины P0LR и P1LR таблиц страниц для P0-P1-областей. Эти регистры загружаются базовыми адресами и длинами таблиц страниц для каждого выполняемого процесса. В отличие от таблицы системных страниц, которая адресуется с помощью физического адреса памяти, обе таблицы страниц для P0- и P1-областей адресуются с помощью виртуальных адресов в системном виртуальном адресном пространстве. В связи с этим для трансляции виртуальных адресов области процесса требуется выполнение дополнительных операций. На первом шаге контроллер памяти с использованием таблицы страниц системы вычисляет физический адрес записи в таблице страниц процесса. Далее трансляция адреса процесса выполняется аналогично трансляции адреса в пространстве системы. Необходимо отметить, что адресация таблиц страниц процесса в виртуальном адресном пространстве системы, а не в физическом связана со сложностью динамического распределения памяти для таблиц страниц в случае адресации непосредственно в физической памяти. Эта сложность определяется тем, что при этом таблицы страниц процесса потребовалось бы размещать в непрерывной физической памяти, т. е. в последовательных страницах. Размер этих таблиц является переменным. Это требование и определяет сложность реализации динамического распределения физической памяти для таблиц страниц процесса.

Схема преобразования виртуального адреса в пространстве процесса (P0-области) представлена на рис. 5.10. В блок-схеме (рис. 5.11) видно отличие процедур трансляции виртуальных адресов в пространствах процесса и системы.

Так как регистры P0BR, P1BR, P0LR и P1LR являются частью аппаратного контекста процесса, то при переключении контекста процесса автоматически осуществляется переключение его адресного пространства.

Число обращений к физической памяти при трансляции виртуальных адресов в физические можно уменьшить за счет аппаратной реализации в СМ1700 механизма сохранения преобразованных адресов в буфере трансляции. Например, при трансляции адреса в пространстве системы требуется одно дополнительное

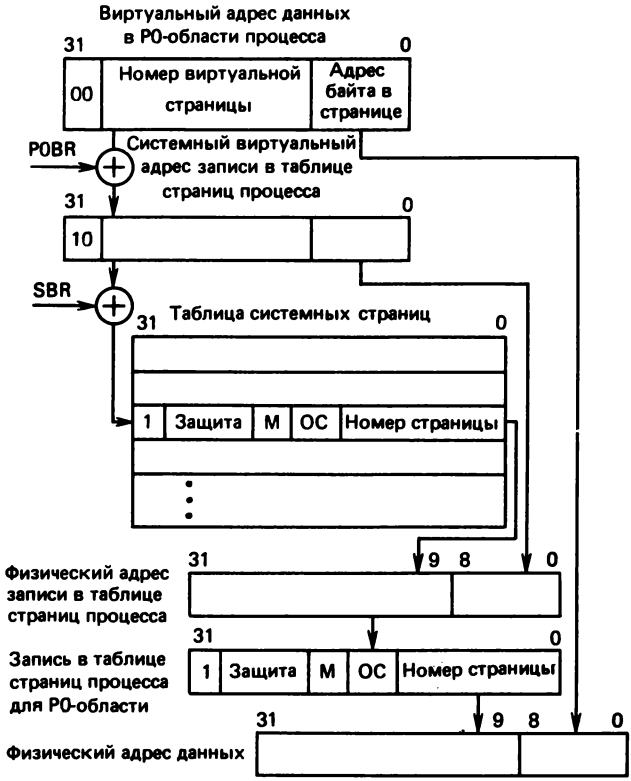


Рис. 5.10. Преобразование виртуального адреса в пространстве процесса

обращение к системной таблице страниц, прежде чем будет получен физический адрес. Таким же образом при трансляции виртуальных адресов в пространстве процесса требуется предварительно выполнить два обращения к физической памяти, что значительно увеличивает накладные расходы системы. Буфер трансляции позволяет их уменьшить при трансляции адресов и особенно эффективно при многократном обращении к одним и тем же виртуальным адресам. Буфер трансляции организован как ассоциативная кэш-память и содержит копии записей в таблице страниц для текущих переадресаций и соответствующие им номера страниц в физической памяти. Всего в буфере трансляции содержится 128 элементов: 64 элемента используются для трансляции в пространстве системы и 64 элемента — в пространстве процесса. Процедура трансляции адресов с использованием буфера осуществляется через поиск в нем элемента, имеющего в качестве метки копию записи в таблице страниц. Если такой элемент найден, то вторая часть этого элемента определяет номер соответствующей страницы в физической памяти. Конкатенация номера страницы и девяти младших разрядов исходного виртуального адреса образует физический адрес.

При переключении контекста процесса часть буфера трансляции, относящаяся к системному адресному пространству, остается неизменной, так как это

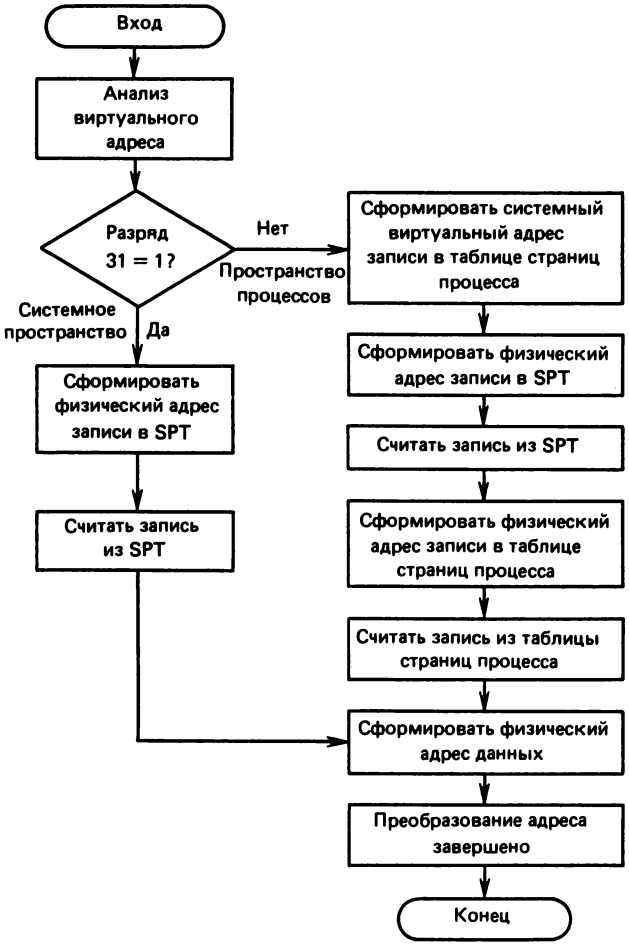


Рис. 5.11. Преобразование виртуальных адресов в пространствах системы и процесса

пространство является общим для всех процессов. Переадресации адресов процесса при переключении контекста процесса становятся недействительными, поскольку они используются только своим процессом.

Привилегированные регистры процессора. Привилегированные регистры в СМ1700 недоступны из программ пользователя и предназначены для использования только операционной системой. К этим регистрам относятся уже рассмотренные выше регистры, используемые для управления памятью. В табл. 5.2 приведен список всех привилегированных регистров в СМ1700, их десятичные номера (процессорные адреса) и тип доступа к ним. Назначение регистров определяется назначением инфор-

Таблица 5.2. **Привилегированные регистры**

Мнемоника	Наименование регистра	Номер	Тип доступа
KSP	Указатель стека режима ядра	0	Ч—з
ESP	Указатель стека режима выполнения	1	Ч—з
SSP	Указатель стека режима супервизора	2	Ч—з
USP	Указатель стека пользовательского режима	3	Ч—з
ISP	Указатель стека прерываний	4	Ч—з
P0BR	Регистр базы таблицы страниц для P0-области	8	Ч—з
P0LR	Регистр длины таблицы страницы для P0-области	9	Ч—з
P1BR	Регистр базы таблицы страниц для P1-области	10	Ч—з
P1LR	Регистр длины таблицы страниц для P1-области	11	Ч—з
SBR	Регистр базы таблицы страниц системной области	12	Ч—з
SLR	Регистр длины таблицы страниц системной области	13	Ч—з
PCBB	Регистр базы блока управления процессом	16	Ч—з
SCBB	Регистр базы блока управления системой	17	Ч—з
IPL	Регистр уровня приоритета прерывания	18	Ч—з
ICCS	Регистр управления и состояния таймера (интервальных часов)	24	Ч—з
NICR	Регистр счета следующего интервала	25	З
ICR	Регистр счета интервала	26	Ч
RXCS	Регистр управления и состояния приемника консоли	32	Ч—з
RXDB	Регистр данных приемника консоли	33	Ч
TXCS	Регистр управления и состояния передатчика консоли	34	Ч—з
TXDB	Регистр данных передатчика консоли	35	З
SID	Регистр идентификации системы	62	Ч
SIRR	Регистр запроса программного прерывания	20	З
SISR	Регистр карты программных прерываний	21	Ч—з

мации, содержащейся в них,—для процессора или для текущего процесса. Чтение и запись привилегированных регистров осуществляется соответственно командам MFPR и MTPR. Привилегированные регистры в этих командах задаются с помощью номеров в соответствии с табл. 5.2. Например, для сохранения значения регистра P1BR и загрузки в него нового значения должны быть использованы следующие команды:

```
MFPR    #10.,R0     ; сохранение P1BR в R0
MTPR    R2,#10.     ; загрузить P1BR новым значением из R2
```

Выполнение операций чтения и записи с привилегированными регистрами разрешено в режиме ядра. Обращение к этим регистрам в других режимах процессора вызывает ошибку. Кроме этого каждый регистр имеет свой тип доступа (ч—з—чтение—запись, ч—чтение, з—запись).

5.3. ПРЕРЫВАНИЯ И НАРУШЕНИЯ

При работе системы некоторые события, возникающие в ней, требуют специального обслуживания. Примером таких событий

являются прерывания от устройств ввода-вывода. Механизм прерываний исключает необходимость периодического опроса устройств для анализа изменения их состояния. Прерывания относятся к событиям, главным по отношению ко всей системе, и обслуживаются в контексте всей системы. Другие события, возникающие при выполнении программы, являются главными по отношению к текущему выполняемому процессу и обслуживаются обычно в контексте этого процесса. Такие события называются особыми ситуациями или нарушениями.

Прерывания. Внешние события, асинхронные по отношению к выполнению текущего процесса. Процессор при прерывании передает управление от текущего процесса специальной программе операционной системы. Так как прерывание не связано с выполняемым процессом, то программа обработки прерывания выполняется с общим для всей системы стеком прерываний. При возникновении прерывания регистры PC и PSL прерванного процесса сохраняются в стеке прерываний, что обеспечивает в дальнейшем после обработки прерывания продолжение выполнения прерванного процесса с использованием команды REI. Так как прерывания возникают асинхронно и несколько устройств ввода-вывода могут выставить запросы на прерывания одновременно, процессор должен иметь механизм приоритетности обслуживания запросов на прерывания. Это означает, что процессор обслуживает всегда запросы на прерывания в соответствии с их приоритетами — сначала с наивысшим приоритетом, затем по степени их приоритетности.

Архитектура СМ1700 обеспечивает обслуживание прерываний на 32 уровнях приоритета, т. е. процессор имеет 32 уровня приоритета прерывания (IPL). Самый низкий приоритет — нулевой — используется для пользовательских программ и большинства обслуживающих программ операционной системы. Следующие 15 уровней (1—15) называются программными и зарезервированы для операционной системы. Остальные 16 старших уровней (16—31) являются аппаратными, используемыми для прерываний от аппаратуры. Уровни от 16 до 23 предназначены для обслуживания внешних устройств и контроллеров, в том числе подключаемых к «Общей шине». Уровни BR4 — BR7 на «Общей шине» в СМ4, СМ1420 соответствуют уровням 20—23 в СМ1700. Уровень 24 используется системным таймером, а уровни от 25 до 31 — для обслуживания ошибочных и аварийных ситуаций в процессоре (отключение питания и др.).

Процессор обслуживает запросы прерывания в соответствии с их приоритетами. Текущий приоритетный уровень процессора (IPL) хранится в разрядах 16—20 в PSL. Если приоритет запроса на прерывание выше, чем текущий IPL процессора, осуществляется обслуживание прерывания. При этом в отличие от 16-разрядных СМ ЭВМ (СМ4, СМ1420 и др.), где вектор прерывания определял

IPL программы обработки прерывания, в СМ1700 процедура обслуживания прерывания осуществляется обычно на приоритетном уровне, определяемом приоритетом запроса прерывания.

Нарушения. Они связаны с возникновением особых ситуаций при выполнении текущей команды и обрабатываются обычно на нулевом уровне IPL в контексте процесса, в котором нарушение произошло. В СМ1700 различают три вида нарушений: ловушки, исправимые и неисправимые ошибки.

Ловушка представляет особую ситуацию, возникающую в конце машинной команды. Некоторые команды явно определяют такие особые ситуации. Например, команды изменения режима доступа процессора всегда вызывают возникновение ловушки и передачу управления системной программе. К процессам пользователей фактически можно отнести два вида ловушек: трассирования, используемые для отладочных целей, и арифметические. Ловушка трассирования возникает между командами и управляется с помощью двух разрядов в PSL. Арифметические ловушки возникают в результате выполнения арифметических операций по следующим причинам:

переполнение в целочисленных, десятичных операциях или операциях с плавающей точкой при получении результата, превышающего формат операнда приемника;

деление на нуль целых, десятичных или чисел с плавающей точкой;

потеря точности в операциях с плавающей точкой при получении результата, не представляемого в формате с плавающей точкой; такой результат записывается нулем.

Так как не всегда желательно выполнение арифметических ловушек, можно запрещать эти ловушки (обнулением соответствующих разрядов PSW).

Нулевой разряд PSW содержит код условия переноса; первый — код условия по переполнению; второй — код условия по нулевому результату; третий — код условия по отрицательному результату; четвертый — разрешение ловушки трассирования; пятый — разрешение ловушки целочисленного переполнения; шестой — разрешение ловушки при потере точности в операциях с плавающей точкой; седьмой — разрешение ловушки десятичного переполнения.

Необходимо отметить, что отсутствует возможность запрещения ловушек при переполнении в операциях с плавающей точкой и при делении на нуль. Кроме этого при вызове процедур с помощью команд CALLG и CALLS можно задать значения разрядов PSW, управляющих ловушками по целочисленному и десятичному переполнению. Этим обеспечивается требуемая реакция вызванной процедуры на эти виды ловушек.

Исправимая ошибка является нарушением, которое возникает при выполнении команды, при этом регистры и память остаются в состоянии, обеспечивающем повторное выполнение команды. Поэтому в счетчике команд PC остается адрес команды, при

выполнении которой произошло это нарушение. Если операционная система может устранить причину нарушения, команда запускается повторно. Примером такого нарушения является обращение к странице памяти, которая отсутствует в физической памяти.

Неисправимая ошибка связана с нарушением, возникающим при выполнении текущей команды, при этом регистры и память остаются в неопределенном состоянии. Поэтому команда не может быть запущена повторно. Примером такой ошибки является недопустимое значение указателя стека ядра при попытке сохранения данных в этом стеке при прерывании или нарушении. Возникающее нарушение приводит к неисправимой ошибке, которая обрабатывается с использованием стека прерываний, так как указатель стека ядра является недопустимым. Дополнительной информации, кроме PSL и PC, при этом в стеке прерываний не сохраняется. Поэтому команда не может быть выполнена снова и операционная система обычно аварийно завершает текущий процесс.

Хотя механизмы обработки прерываний и нарушений всех трех видов сводятся к переключению процессора на специальную системную программу обработки этих событий в системе, между ними имеются существенные различия:

1. Нарушения вызываются командами выполняемого процесса, т. е. являются синхронными по отношению к выполняемому процессу. Прерывания асинхронны по отношению к выполняемому процессу, так как связаны с внешними по отношению к процессу событиями.

2. Нарушения обычно обрабатываются в контексте процесса, который привел к нарушению, и выполняются с использованием стека, определяемого текущим режимом процесса. Прерывания обслуживаются независимо от текущего выполняемого процесса, т. е. в контексте операционной системы с использованием стека прерываний.

3. Уровень приоритета прерывания процессора (IPL) при нарушениях обычно не изменяется, в то время как при инициации прерывания IPL всегда повышается до уровня приоритета источника прерывания (устройства ввода-вывода, контроллера и др.).

4. Разрешенные нарушения всегда инициируются независимо от уровня IPL процессора. Обработка прерываний откладывается до момента, когда уровень IPL процессора станет ниже приоритета запроса прерывания.

5. Большинство нарушений не может быть запрещено. Если условие нарушения наступает при его запрете, то инициации не произойдет и при отмене запрета. Запрос на прерывание, имеющий приоритет ниже или равный IPL процессора или возникающий в момент, когда это прерывание запрещено, ставится в очередь на обслуживание. При возникновении условий прерывания, если запрос на него еще существует, данное прерывание будет удовлетворено.

6. Поле предыдущего режима в PSL при прерывании всегда устанавливается в режим ядра, а при нарушениях — в режим, существовавший в этот момент.

Векторы прерываний. Передача управления программе обработки прерывания или нарушения осуществляется с использованием вектора, который является двойным словом, определяющим адрес программы обработки и способ обслуживания прерываний и нарушений. Векторы для всех видов нарушений, программных и аппаратных прерываний содержатся в системной структуре данных, называемой блоком управления системой (SCB). Физический адрес SCB содержится в привилегированном регистре базы блока управления системой SCBB. Размещение векторов в SCB представлено в табл. 5.3. Каждый вектор интерпретируется аппаратурой следующим образом.

Таблица 5.3. **Блок управления системой**

Вектор	Назначение
	При нарушениях
04	Схемный контроль
08	Недостоверный стек ядра
0C	Неисправность питания
10	Резервная (привилегированная) инструкция
14	Инструкция XFC
18	Недопустимый операнд
1C	Недопустимый режим адресации
20	Нарушение режима доступа
24	Ошибка преобразования адреса
28	Ловушка трассирования
2C	Инструкция BPT
30	Нарушение режима совместимости
34	Арифметическая ловушка
⋮	Не используются
40	Переход в режим ядра (CHMK)
44	Переход в режим выполнения (CHME)
48	Переход в режим супервизора (CHMS)
4C	Переход в режим пользователя (CHMU)
⋮	Не используются
	При прерываниях
84	Программный уровень 1
88	Программный уровень 2
8C	Программный уровень 3
	Программные уровни 4 — E
BC	Программный уровень F
C0	Системный таймер
⋮	Не используются
100	Векторы прерываний устройств
⋮	
1FF	»

Разряды 1 и 0 содержат код, определяющий, как должно обрабатываться событие, связанное с прерыванием или нарушением:

00 — определяет обработку события через стек ядра; если процессор уже работает со стеком прерываний, то он используется и для обработки события. Разряды 31 ... 2 вектора определяют виртуальный адрес программы обслуживания события. Так как этот адрес должен быть на границе двойного слова, то виртуальный адрес формируется полностью добавлением к этим разрядам двух младших нулевых разрядов;

01 — определяет обработку события через стек прерываний, при этом IPL процессора повышается до 31. Разряды 31 ... 2 вектора определяют виртуальный адрес программы обслуживания события;

10 — предназначен для событий, обрабатываемых с использованием микропрограмм в перезаписываемой управляющей памяти (WCS). Разряды 15 ... 2 определяют адрес микропрограммы;

11 — не используется.

Программные прерывания. В СМ1700 предусмотрены 15 уровней программных прерываний. Так как каждый приоритетный уровень прерывания IPL имеет свой вектор прерывания, операционная система может использовать программные прерывания как вызовы системных программ, выполняющих определенные функции. Программа, выполняемая в режиме ядра, может вызвать системную программу, выставляя запрос на прерывание на соответствующем уровне IPL. Наличие нескольких программных уровней приоритетного прерывания позволяет операционной системе разделять и синхронизировать выполнение разных функций. Например, переключение контекста процесса осуществляется всегда на третьем уровне, что гарантирует при работе операционной системы невозможность переключения контекста на любом другом уровне приоритетного прерывания.

Для управления программными прерываниями используются три привилегированных регистра. Регистр карты программных прерываний (SISR) содержит общую сводку запросов на программное прерывание для всех 15 приоритетных уровней этих прерываний. Единичные состояния разрядов 0 ... 15 в этом регистре определяют наличие запросов на программное прерывание на соответствующих уровнях приоритета. Когда уровень IPL процессора окажется ниже приоритета одного из запросов программного прерывания, немедленно генерируется прерывание на уровне IPL процессора, соответствующего приоритету запроса. Регистр доступен для чтения и записи с помощью команд MFPR и MTPR. Запись непосредственно в регистр SISR обычно используется только при запуске и рестарте системы. Появление в этом регистре запросов на прерывание обычно вызывается через регистр SIRR.

Регистр запроса программного прерывания (SIRR) является привилегированным и используется для запроса программного прерывания на одном из 15 уровней приоритета. Запрос на прерывание вызывается операционной системой с помощью команды MTPR, по которой в SIRR записывается требуемое значение уровня прерывания (приоритета запроса на прерывание) в пределах от 1 до 15. Если требуемый уровень прерывания окажется ниже или равным текущему уровню IPL процессора, обслуживание запроса на программное прерывание откладывается до тех пор, пока IPL процессора не станет ниже приоритета запроса на прерывание. При этом в регистре SISR будет установлено единичное значение разряда, соответствующее приоритету запроса. Если приоритет запроса выше уровня IPL процессора, немедленно вырабатывается прерывание.

Регистр приоритетного уровня прерывания IPL используется для установки требуемого значения приоритета процессора в PSL (разряды 20... 16). Системная программа, выполняемая в режиме ядра, имеет возможность доступа к этому регистру по командам MTPR и MFPR. По команде MFPR можно повысить или понизить уровень приоритета процессора. При понижении текущего уровня приоритета процессора может произойти прерывание от ожидающих запросов на прерывание в регистре SISR.

Глава 6. СТРУКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ МОС ВП

6.1. ОСНОВНЫЕ КОМПОНЕНТЫ МОС ВП

На рис. 6.1 схематически представлена архитектура СМ1700 с многофункциональной операционной системой с виртуальной памятью (МОС ВП). Из рисунка видно, что такую архитектуру можно представить как многоуровневую: аппаратные средства, программно-аппаратные и, наконец, программные. С точки зрения вопросов, интересующих нас в данный момент, вычислительную систему следует рассматривать как устройство, имеющее целый ряд интерфейсов для управления самим собой. Осуществляет такие связи пользователей и их программ с аппаратурой ЭВМ операционная система. Рассмотрим, какие интерфейсы предоставляет пользователю МОС ВП и как они реализуются.

Системные обслуживающие процедуры. Если вновь обратиться к рис. 6.1, то можно увидеть, что первым слоем МОС ВП, непосредственно связанным с устройствами, является уровень

Рис. 6.1. Программно-аппаратная многоуровневая архитектура вычислительной системы

процедур ядра. Процедуры реализуют все функции, затребованные у системы с использованием соответствующих интерфейсов. На этом уровне система предоставляет следующие основные услуги:

Обслуживание флагов событий. Так же, как и в операционной системе реального времени ОС РВ [4], в МОС ВП для синхронизации процессов с различными событиями в системе и вне ее применяются флаги событий. Этот механизм позволяет маркировать события, важные не только для одного конкретного процесса, но и для некоторой их группы или даже для всех известных системе процессов.

Обработка асинхронных системных прерываний (AST-прерываний). В МОС ВП выполнение процесса может быть прервано некоторым собитием для того, чтобы отработала специальная AST-программа, ассоциированная с таким событием. Аналогично ОС РВ, МОС ВП позволяет пользователю работать с такими прерываниями с помощью соответствующего механизма.

Работа с логическими именами устройств. Система МОС ВП обеспечивает возможность создания программ, инвариантных по отношению к устройствам ввода-вывода, в результате назначения реальным устройствам логических имен.

Выполнение операций ввода-вывода. В системе МОС ВП процедуры ввода-вывода обеспечивают обмен с внешними устройствами на уровне физических, логических и виртуальных блоков или записей. Кроме того, они обеспечивают передачу сообщений системным процессам, а при использовании специальных сетевых средств — обмен данными с процессами, функционирующими в удаленных друг от друга СМ1700.

Управление процессами. Группа услуг, включающая создание процессов, слежение за их функционированием и их уничтожение.

Обслуживание таймера. Процессы МОС ВП могут заказывать прерывания в требуемые моменты, а также имеют доступ к текущим дате и времени, в том числе и в преобразованном печатном формате.

Обработка исключительных ситуаций. Процедуры получают управление в том случае, когда возникает программная или аппаратная исключительная ситуация во время выполнения процесса.

Управление памятью. Служба, позволяющая процессу управлять своей виртуальной и (или) физической памятью.

Изменение режима. Если некоторый процесс обладает соответствующими привилегиями, он может затребовать изменение режима доступа к процессору для выполнения некоторой программы.

Обращение к любой такой процедуре или, иначе, средству системного обслуживания осуществляется через интерфейс системного обслуживания, представляющий собой набор примитивов, реализованных в виде макрокоманд (для программ на языке ассемблера) или подпрограмм (для программ на языках высокого уровня). В последующих разделах данной главы наиболее важные из перечисленных системных средств будут обсуждены подробнее.

Файловая система и логический ввод-вывод. Следующий слой приведенной на рис. 6.1 архитектуры содержит компоненты, обеспечивающие доступ пользователя и его программ к данным, хранящимся в файлах, расположенных на структурированных носителях. Мы специально подчеркиваем эту логическую структурированность, поскольку обмен информацией с устройствами, не имеющими файловой организации (или когда необходимо игнорировать ее), выполняется службой ввода-вывода, реализованной в процедурах ядра.

Таким образом, в рассматриваемом слое создается и обслуживается файловая структура носителей СМ1700 и обеспечивается логический доступ к хранящимся на них данным. Все эти функции реализует система управления данными RMS, которая оперирует как целыми файлами, так и отдельными хранящимися в них записями. Система RMS поддерживает три файловых организации: последовательную, индексную и относительную. Для обеспечения возможностей работы с файлами, имеющими различную организацию, в системе управления данными реализованы три базовых метода доступа: последовательный, произвольный и адресный. При необходимости (для оптимизации работы с данными) пользователь может применять динамический режим, который допускает переключение с одного базового метода доступа на другой во время функционирования процесса.

На уровне файлов пользовательский процесс может выполнить одну или несколько операций из следующего набора: создать

файл; открыть существующий файл; модифицировать атрибуты файла; расширить файл; закрыть файл; стереть файл. При работе с записями RMS допускает ввод, вывод, изменение и стирание.

Взаимодействие пользовательских процессов с системой RMS осуществляется с помощью примитивов, которые, как и для обслуживающих процедур ядра, реализованы в виде макрокоманд и подпрограмм.

Командный язык. Следующий предоставляемый системой МОС ВП слой представляет собой командный язык оператора системы. Для интерпретации вводимых человеком команд и запуска требуемых операций на выполнение в состав МОС ВП входит интерпретатор DCL. Командный язык МОС ВП включает инструкции, позволяющие создавать необходимую для процесса программную среду и выполнять другие функции управления. Не будучи в полном смысле языком программирования, он, тем не менее, содержит целый ряд команд, позволяющих выполнять требуемые обстановкой действия без участия человека в результате использования специальных файлов (командных процедур), хранящих набор фраз командного языка. Наиболее употребительные из команд оператора МОС ВП будут рассмотрены в 6.7.

Средства программирования и средства расширения функциональных возможностей системы. Очередной слой МОС ВП представлен языками систем программирования и другими средствами, необходимыми для создания новых программ. Сюда же относятся программы, обеспечивающие расширение функциональных возможностей МОС ВП. Рассмотрим кратко основные компоненты этого слоя.

Языки программирования. Система МОС ВП включает значительное число языковых средств, в том числе макроассемблер, Фортран и Фортран-77, Кобол, Паскаль, Бейсик, ПЛ/1. Ожидается в дальнейшем появление в МОС ВП языков СИ, Блисс и Корал.

Текстовые редакторы. Для обеспечения работы с различными текстами, в том числе с исходными текстами программ, МОС ВП предоставляет пользователю строчный интерактивный редактор SOS, экранный — EDT и пакетный — SLP.

Компоновщик LINKER выполняет компоновку различных объектных модулей и элементов объектных библиотек в единую программную структуру, а также запись ее в виде образа процесса в файл.

Отладчик системы МОС ВП обеспечивает возможность интерактивной отладки программ в символьном виде. Это допустимо для большинства языковых процессоров: Фортрана, Кобола, Бейсика и макроассемблера.

Библиотекарь МОС ВП позволяет создавать, модифицировать, распечатывать и обслуживать библиотеки макрокоманд, объектных модулей и разделяемых образов. В качестве последних

в МОС ВП могут выступать разделяемые резидентные библиотеки и разделяемые общие области.

Общая библиотека исполнительных процедур RTL МОС ВП представляет собой набор подпрограмм как общего применения, так и ориентированных на определенные языки. Эти процедуры предназначены для формирования общей исполнительной среды для пользовательских программ, написанных на различных языках программирования. Библиотека RTL содержит следующие функциональные секции: распределения ресурсов (виртуальной памяти, логических номеров устройств, флагов событий); обработки исключительных ситуаций; утилит общего назначения; математических методов. Кроме того, RTL включает также специфичные для каждого языка программирования средства обработки ошибок и управления данными.

Дополнительные программные средства. Под этим весьма условным названием объединены средства взаимодействия процессов между собой. К ним относятся:

Система программного обеспечения распределенных сетей — ТРАЛ. Обеспечивает возможность построения различных сетевых конфигураций на базе СМ1700, СМ1420 и других СМ ЭВМ.

Система программного обеспечения локальных сетей — МАГИСТР. Позволяет объединять машины СМ1700 в локальную высокоскоростную сеть магистрального типа, соответствующую стандартам Ethernet.

Система программного обеспечения для организации неоднородных комплексов на базе СМ1700 и ЕС ЭВМ — ЭМУЛЯТОР. Предназначена для создания неоднородных многомашинных систем высокой производительности.

Средства электронной почты МОС ВП. Обеспечивают обмен сообщениями как между терминалами одной или нескольких ЭВМ, так и между процессами. Надо подчеркнуть, что в последнем случае процессы должны использовать соответствующие системные обслуживающие процедуры.

Реконфигурация и тестирование. Система МОС ВП снабжена реконфигуратором, обеспечивающим ее настройку на конфигурацию конкретной ЭВМ СМ1700. Кроме того, система включает большое число диагностических процедур, предназначенных для тестирования аппаратуры и поиска ошибок.

На этом мы завершим анализ компонентов системы МОС ВП и реализуемых ими функций, а также их интерфейсов. Если не считать диагностических процедур, многие из которых являются микропрограммными, то можно говорить, что в основе большинства средств лежит использование процесса — основной программной структуры, которой оперирует МОС ВП. Более того, именно на уровне процессов применяются два из уже рассмотренных нами типов интерфейсов: интерфейс системного обслуживания и интерфейс системы управления данными. По-

этому теперь мы перейдем к обсуждению функционирования процессов в операционной системе МОС ВП.

6.2. РАСПРЕДЕЛЕНИЕ ВРЕМЕНИ ПРОЦЕССОРА. ПЛАНИРОВАНИЕ ПРОЦЕССОВ

Ранее мы уже говорили об аппаратной реализации управления процессами в СМ1700 и кратно рассмотрели планировщик, распределяющий между ними время центрального процессора. Для более глубокого понимания принципов планирования процессов в МОС ВП и реализующих их механизмов в данном разделе основное внимание будет уделено анализу алгоритмов, применяемых для диспетчеризации. Поскольку речь идет о МОС ВП, то рассматриваются лишь те алгоритмы, которые на этапе проектирования МОС ВП исследовались на возможность применения в ней.

Алгоритмы планирования процессов. В системах, подобных МОС ВП, одна из важнейших задач планировщика заключается в такой организации последовательного запуска готовых к выполнению процессов, при которой обеспечивается наиболее эффективное обслуживание пользователей в смысле некоторого критерия: это может быть равномерное обслуживание, первоочередное обслуживание пользователей, ориентированных на работу в реальном масштабе времени, и т. д. В настоящее время существует большое число стратегий, с бóльшим или меньшим успехом приводящих к решению сформулированной задачи. Здесь можно говорить о схемах, основанных на приоритетах, учете размера процесса, оценке его месторасположения в памяти или готовности к выполнению. В разных системах возможны различия в функционировании планировщиков, обусловленные особенностями процессов и их зависимостью от среды. Например, в одних операционных системах предпочтение отдается интерактивным процессам, в других — программам вычислительного характера, выполняемым в пакетном режиме, системы третьего типа ориентируются в первую очередь на задачи, критичные по времени к наступлению некоторых событий.

При использовании любой стратегии планирования система оперирует элементами *очередей*, т. е. некоторыми таблицами, описывающими каждый планируемый к выполнению процесс. Именно планировщик упорядочивает эти элементы в соответствии с принятой стратегией или дисциплиной обслуживания. Так, при реализации одной из простейших из них формируется общая очередь выполнимых процессов, а планирование их выполнения производится по *циклическому алгоритму*. В соответствии с рассматриваемой схемой каждый раз, как только процессор становится доступным, для выполнения выбирается тот процесс, описывающая таблица которого находится в начале очереди. После истечения отведенного процессу *кванта* (интервала) времени

владения процессором планировщик помещает соответствующий элемент в конец очереди, выбирает из ее начала первый и по нему запускает новый процесс. Именно таким образом осуществляется периодический запуск каждого готового к выполнению процесса с предоставлением всем им равных возможностей использования времени центрального процессора (при условии, что всем им выделяется одинаковый квант времени).

Если в операционной системе определенным процессам или классам процессов должно отдаваться предпочтение, то целесообразно использование *приоритетной схемы* выборки. Процессы, обладающие более высокими приоритетами, получают процессорное время раньше, нежели менее приоритетные. В таких алгоритмах применяются различные методы задания приоритетов. Например, часто используется правило, в соответствии с которым каждому пользователю присваивается некоторый начальный приоритет, значение которого определяется классом задания данного пользователя. Этот исходный приоритет может быть в дальнейшем модифицирован в зависимости от активности обрабатываемого процесса. В соответствии с другими методами значения приоритетов задаются согласно прогнозируемым временам центрального процессора, необходимым для завершения выполнения процессов. Базирующиеся на алгоритмах прогнозирования планировщики, выбирая задания из очереди, отдают предпочтение тем из них, которые должны завершиться быстрее других. Подобные стратегии заставляют пользователей следить за сбалансированностью потребностей их программ в ресурсах системы. Если говорить о планировщиках операционных систем разделения времени, то следует отметить их общую особенность: независимо от применяемой приоритетной стратегии каждый раз, как только истекает квант времени, выделенный текущему процессу, выполнение последнего приостанавливается и запускается следующий процесс из очереди.

Прежде чем перейти к рассмотрению планировщика системы МОС ВП, обсудим специальную управляющую структуру данных, называемую *программным блоком управления процессом* PCB. Такой PCB содержит всю программную контекстную информацию о процессе. Как аппаратный управляющий блок хранит информацию об аппаратных средствах, необходимую для управления процессом, так и программный PCB объединяет данные, относящиеся к контексту, определяющему процесс как структуру, существующую в среде МОС ВП. В частности, программный PCB содержит описание привилегий пользователя, его имя, в нем накапливается информация об использованных ресурсах и т. д. Кроме того, такой блок содержит ссылку на соответствующий аппаратный PCB, содержимое которого необходимо для формирования контекстного ключа при первичном или повторном запуске процесса.

Именно программные PCB являются элементами очередей планировщика МОС ВП. Планировщик отслеживает состояние

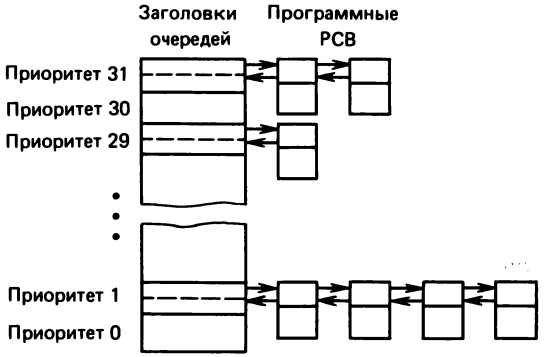


Рис. 6.2. Заголовки очередей выполняемых процессов

каждого процесса на основе результатов анализа содержимого этих очередей. Он же с учетом состояний процессов и значений их приоритетов формирует очереди из таких управляющих структур. В любой момент времени каждый известный системе процесс описывается блоком PCB, обязательно включенным в одну или более очередей планировщика.

Выбор очередного процесса для выполнения в МОС ВП производится достаточно просто. Каждому из них присвоен некоторый приоритет от 0 до 31. Как только размещенный в памяти процесс оказывается готовым для выполнения, он помещается в одну из 32 создаваемых планировщиком приоритетных очередей выполняемых процессов (рис. 6.2). При выборке планировщик обращается к началу той из непустых очередей, которая обладает наивысшим приоритетом.

При недостаточно глубоком рассмотрении подобного механизма выбора может сложиться впечатление, что высокоприоритетные процессы исключают возможность доступа к процессору низкоприоритетных. Такое представление не соответствует истине, поскольку именно во избежание этого планировщик осуществляет динамическое изменение значений приоритетов. Так, при создании процесса в момент запуска задания он получает приоритет, назначенный системным администратором соответствующему пользователю. В дальнейшем это значение может быть увеличено или уменьшено в зависимости от активности процесса. Например, если полностью исчерпан выделенный процессу квант времени, то его приоритет уменьшается и соответствующий программный PCB помещается в конец менее приоритетной, чем прежде, очереди. Другой случай: завершается затребованный процессом ввод-вывод. Если операция была синхронной (т. е. процесс ожидал завершения обмена), то значение приоритета повышается и процесс получает возможность быстрее перейти в активное состояние. Отметим одну важную особенность механизма диспетчеризации,

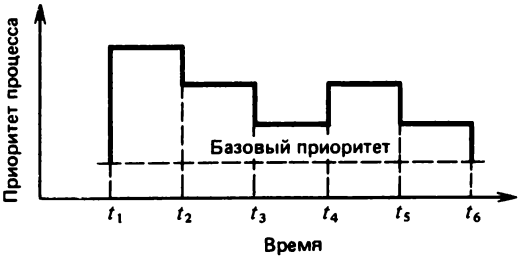


Рис. 6.3. Изменение приоритета процесса во времени

реализованного в МОС ВП: в любом случае приоритет процесса никогда не может стать ниже своего начального значения.

К изменениям приоритетов в МОС ВП могут привести следующие события:

истечение кванта времени (приоритет уменьшается);

завершение терминального ввода (приоритет повышается);

завершение терминального вывода (приоритет повышается);

завершение обмена с другими внешними устройствами или отказ при поиске страницы в рабочем наборе (приоритет повышается);

освобождение ресурсов (приоритет повышается).

По мере выполнения процесса его приоритет изменяется каждый раз, когда происходит одно из перечисленных событий. На рис. 6.3 представлена диаграмма возможных изменений приоритета некоторого процесса во времени. Если подвергнуть анализу приведенный выше список событий, то становится очевидным, что в МОС ВП предпочтение отдается интерактивным программам. Это проявляется в повышении приоритетов процессов сразу после окончания диалога с пользователем (окончания терминального ввода или вывода) и в снижении их значений после того, как процесс приобретет черты вычислительного (истечение кванта времени). Наибольшее изменение приоритетов происходит после завершения терминального ввода.

В системе МОС ВП выделяют два основных типа процессов: разделения времени (диалоговые) и реального времени. С точки зрения рассматриваемых нами вопросов они различаются в первую очередь принципами их диспетчеризации. Так, базовые значения приоритетов процессов разделения времени лежат в интервале от 0 до 15, а в случае реального времени — от 16 до 32. Именно в этих интервалах могут изменяться значения приоритетов выполняемых процессов обоих типов. Реальное время, кроме того, связано с обработкой быстро чередующихся событий, в результате чего соответствующие процессы не теряют контроль над центральным процессором, пока не завершатся по собственной инициативе, не перейдут в ожидание завершения операции ввода-вывода или пока не будут прерваны однотипным более приоритетным процессом. В отличие от них процессы разделения времени прерываются по истечении интервала времени владения процессором.

Состояние процесса. Мы будем говорить, что состояние представляет собой часть контекста процесса, описывающую условия его существования. Так, если процесс размещен в памяти и может выполняться, то он закрепляется за одной из 32 очередей, соответствующих состоянию, которое условно можно назвать «выполняемый и резидентный». Другие 32 очереди создаются для процессов, которые могут выполняться, но временно удалены из памяти.

Кроме выполняемых в системе обычно существуют процессы, которые по различным причинам в данный момент не могут выполняться, поскольку ожидают возникновения тех или иных событий, например завершения операции ввода-вывода, обработки страничного отказа, освобождения или готовности программных и (или) аппаратных ресурсов и т. д. Все они фиксируются в одной из 11 очередей ожидания, создаваемых и реорганизуемых планировщиком (табл. 6.1). Как только происходит событие, ожидаемое некоторым процессом, внутри очереди перемещается его программное PCB, т. е. отражается изменение его состояния. Размещение процессов, находящихся в различных состояниях в разных очередях, позволяет уменьшить затраты на поиск ждущего процесса в момент возникновения события, необходимого для его активизации. Например, если некоторый процесс ожидает завершения обмена страницы, то его программный PCB находится в соответствующей очереди и, как только требуемая страница будет найдена и введена в рабочий набор, планировщик просматривает эту очередь ожидания, отыскивает PCB такого процесса и в зависимости от приоритета включает его в одну из очередей выполняемых процессов. Выбор очереди определяется и тем, резидентен процесс в памяти или нет. На рис. 6.4 представлен граф состояний процессов в операционной системе МОС ВП. Стрелками обозначены возможные переходы из одного состояния в другое.

После завершения обработки любого события планировщик МОС ВП делает попытку выполнить выравнивание приоритетов различных процессов, для чего останавливает те из них, которые работают в течение длительного времени, и дает возможность занять процессор остальным. При этом наибольшее предпочтение отдается интерактивным программам с высокой частотой обмена с пользователем. Вывод процесса из активного состояния сопровождается сохранением всей необходимой в дальнейшем информации о нем в системных структурах данных аппаратных и программных PCB для того, чтобы впоследствии возобновить его выполнение.

Итак, планировщик МОС ВП осуществляет выбор очередного процесса на выполнение, а также выполняет ряд действий, связанных с реакцией на изменение состояния любого процесса, известного системе, создает и поддерживает соответствующие очереди. Из этих функций особый интерес для нас представляет

Таблица 6.1. **Очереди ожидания невыполняемых процессов в системе МОС ВП**

Номер очереди	Состояние процесса	Описание состояния
1	Ожидание обмена страницы	Процессу не удалось завершить передачу страницы (при чтении ее в память или записи на диск)
2	Ожидание общего флага	Состояние ожидания установки общего флага событий [1]
3	Ожидание свободной страницы	Состояние ожидания свободной страницы оперативной памяти, необходимой для выполнения процесса
4	Ожидание перевода в неактивное состояние	Процесс выдал запрос на перевод в неактивное состояние, но находится в памяти
5	Ожидание перевода в неактивное состояние с выгрузкой	Процесс выдал запрос на перевод в неактивное состояние и выгружен из памяти
6	Ожидание локального флага	Ожидание установки локального флага событий [2], процесс резидентен в оперативной памяти
7	Ожидание локального флага с выгрузкой	Ожидание установки локального флага событий, процесс выгружен из оперативной памяти
8	Ожидание после приостановки выполнения	Процесс размещен в памяти, но его выполнение приостановлено
9	Приостановлен и выгружен	Выполнение процесса приостановлено, и он выгружен из оперативной памяти
10	Ожидание ресурса	Ожидание освобождения системных ресурсов
11	Ожидание обслуживания по страничному отказу	Ожидание считывания страницы после отказа при обращении к ней

[1] Общие флаги события используются для индентификации того, что произошло событие, имеющее значение для всех процессов системы или некоторой их группы.
[2] Локальные флаги идентифицируют наступление событий, важных для каждого процесса.

переключение центрального процесса с выполнения одного процесса на выполнение другого, т. е. механизм контекстного переключения. Для более глубокого понимания того, как в МОС ВП осуществляется диспетчеризация, обсудим планировщик подробнее.

Выполнение контекстного переключения. Ранее мы уже рассматривали, как при переключении контекста используется содержимое аппаратного РСВ. Попытаемся теперь понять, что происходит в этом случае с программными РСВ и как в целом реализовано такое переключение.

Контекстное переключение в МОС ВП осуществляет один из компонентов планировщика — *перепланировщик*. Эта сервисная процедура переключения процессора с контекста прерываемого процесса на контекст вновь запускаемого реализована с использованием программных прерываний различного уровня или приоритета для вызова операционной системы. Иными словами, запуск процедуры можно рассматривать как реакцию системы на программное прерывание. Делается это следующим образом.

В системе функционируют процедуры, обнаруживающие изменения в состояниях процессов или истечение квантов времени.

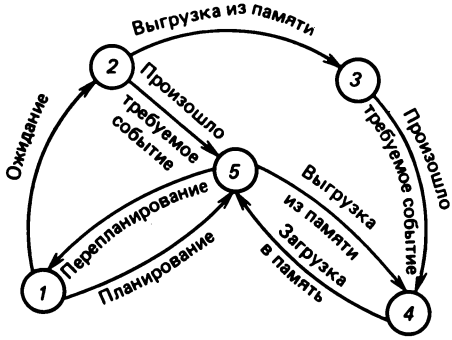


Рис. 6.4. Изменение состояний процессов в МОС ВП:

1 — в настоящий момент выполняется; *2* — находится в очереди ожидания, соответствующей причине прекращения активности; *3* — аналогично *2*, но процесс выгружен на диск; *4* — находится в очереди выполнимых, но не загруженных в память процессов; *5* — находится в очереди выполнимых и загруженных в память процессов

Они работают с начальным значением программного приоритета, большим 3. Если истекает выделенный процессу квант времени или происшедшее событие приводит к тому, что некоторому процессу назначается приоритет, больший, чем у выполняемого, или более приоритетный процесс становится активным, то должна осуществляться инициация контекстного переключателя. Это реализует программа регистрации событий простым обращением к системе с требованием установки программных прерываний на уровень 3, т. е. на тот, на котором работает перепланировщик. После регистрации события, в момент попытки перейти к приоритетному уровню ниже 3, запускается перепланировщик и выполняется переключение контекста.

Листинг программы-перепланировщика, включенного в операционную систему МОС ВП, приведен на рис. 6.5. В целом эта программа достаточно проста, хотя на первый взгляд может показаться сложной из-за использованных в ней весьма длинных символических имен. Часто применяемая в программе запись вида W ∧ symbol предназначена для указания ассемблеру сгенерировать относительные адреса слов. (По умолчанию для переменных, не определенных в транслируемом модуле, ассемблер использует относительную адресацию с длинными словами. Подобная адресация позволяет сэкономить память для переменных, которые находятся в пределах непосредственной достижимости.) Для понимания того, что именно делает приведенная процедура, мы должны предварительно определить следующие используемые в тексте обозначения:

1. SCH$GL_COMQS — адрес 32-разрядного длинного слова состояния вычисляемой очереди COMQS[1]. Каждый бит в COMQS представляет одну из 32 очередей выполняемых процессов, показанных на рис. 6.2. В практических реализациях как очереди,

[1] Вычисляемой здесь и далее называется очередь, описанная таким образом, что ею можно оперировать с использованием средств, предоставляемых архитектурой CM1700 (команды FFS, INSQUE, REMQUE и т. д.).

```
;++++
;SCH$RESHED - Обработка прерываний.
;Вход в эту подпрограмму осуществляется посредством
;прерывания по переиспетчеризации (IPL 3).
;
;Среда выполнения:
;IPL=3; Режим = KERNEL; IS=0
;Вход:
;       00(SP)=PC    В момент прерывания
;       04(SP)=PSL   В момент прерывания
;++++
SCH$RESHED::
      SETIPL  #IPL$_SYNCH              ;Синхронизация с регистратором
      SVPCTX                           ;Сохранить контекст процесса
      MOVL    W^SCH$GL_CURPCB,R1       ;Получить адрес текущего PCB
      MOVZBL  PCB$B_PRI(R1),R2         ;Определить текущий приоритет
      BBSS    R2,W^SCH$GL_COMQS,10$    ;Очередь не пуста
10$:  MOVW    #SCH$C_COM,PCB$W_STATE(R1) ;Установить состояние "RES"
      MOVAQ   W^SCH$AQ_COMT[R2],R3     ;Вычислить адрес начала очереди
      INSQUE  (R1),@(R3)+              ;Поместить в конец очереди
;++
;SCH$SCHED - Планирование выполнения процесса
;
;Эта подпрограмма осуществляет выбор наиболее приоритетного
;выполнимого процесса и подготавливает его запуск
;++
SCH$SCHED::                              ;Планирование выполнения
      SETIPL  #IPL$_SYNC               ;Синхронизация с регистратором
      FFS     #0,#32,W^SCH$GL_COMQS,R2 ;Поиск процесса
      BEQL    SCH$IDLE                 ;Нет готового процесса
      MOVAQ   W^SCH$AQ_COMH[R2],R3     ;Вычислить начало очереди
      REMQUE  @(R3)+,R4                ;Выборка заголовка
      BVS     QEMPTY                   ;Переход при пустой очереди
                                       ;(фатальная ошибка)
      BNEQ    20$                      ;В очереди больше одного элемента
      BBCC    R2,W^SCH$GL_COMQS,20$    ;Установка признака
                                       ;"Очередь пуста"
20$:  CMPB    #DYN$C_PCB,PCB$B_TYPE(R4) ;Должен быть PCB
      BNEQ    QEMPTY                   ;Переход, если не PCB
                                       ;(фатальная ошибка)
      MOVW    #SCH$C_CUR,PCB$W$STATE(R4) ;Установить текущее
                                       ;состояние
      MOVL    R4,W^SCH$GL_CURPCB       ;Зафиксировать текущее PCB
      CMPB    PCB$B_PRIB(R4),PCB$B_PRI(R4) ;Базовый приоритет
      BEQL    30$                      ;Да, не менять приоритет
      BBC     $4,PCB$B_PRI(R4),30$     ;Не менять приоритет
                                       ;реального времени
      INCB    PCB$B_PRI(R4)            ;Вернуться к базовому
                                       ;приоритету
30$:  MOVB    PCB$B_PRI(R4),W^SCH$GB$_PRI ;Установить глобальный
                                        ;приоритет
      MTPR    PCB$L_PHYPCB(R4),#PR$_PCBB ;Восстановить физический
                                       ;адрес PCB
      LDPCTX                           ;Восстановить контекст
      REI                              ;Нормальный возврат
```

Рис. 6.5. Программа перепланировщика МОС ВП

так и биты COMQS упорядочены в направлении от высшего приоритета к низшему. Следовательно, бит 0 в COMQS и очередь 0 представляют процессы с наивысшим приоритетом, равным 31. Для поиска установленных битов в поле удобно использовать

команду FFS («Поиск первого установленного бита»). Именно она включена в рассматриваемую программу для определения установленного бита COMQS, соответствующего очереди наиболее приоритетных процессов. (В этом случае, когда некоторый бит длинного слова состояния очереди сброшен, представляемая им очередь процессов пуста.)

2. SCH$AQ_COMH — адрес заголовка первой из 32 вычисляемых очередей (см. рис. 6.2). Полученный из SCH$GL_COMQS номер установленного бита используется в качестве индекса при вычислении адреса заголовка соответствующей ему приоритетной очереди.

3. SCH$AQ_COMT — адрес указателя последнего элемента первой очереди (адрес 2-го длинного слова ее заголовка).

4. SCH$GL_CURPCB — адрес указателя программного блока управления процесса, выполняющегося в текущий момент.

Более детально разберем отдельные фрагменты приведенной на листинге программы-перепланировщика. Рассмотрим, что делает каждый из них

1. Процедура запускается в результате прерывания типа IPL3. Затем она получает более высокий приоритетный уровень, соответствующий уровню синхронизации в системе МОС ВП (IPL7, представленный символически как IPL$_SYNC), что позволяет обеспечить ее синхронизацию с регистрацией событий. Этим достигается уверенность, что ни одно изменение состояний процессов или данных не сможет произойти в период выполнения перепланирования (см. строки 12—14).

2. Программа сохраняет адрес программного PCB текущего процесса в регистре R1, а значение его приоритета заносится в регистр R2. Блок управления процессом помещается в конец очереди для данного приоритета. Соответствующий бит в длинном слове состояния вычисляемой очереди устанавливается для обозначения того, что процесс поставлен в очередь (строки 15—20).

3. Команда FFS выделяет первый установленный в единицу бит в длинном слове состояния вычисляемой очереди. Таким образом определяется непустая очередь с наивысшим приоритетом. Вычисляется адрес ее заголовка и производится перемещение программного PCB. Этот блок соответствует новому предназначенному для обработки процессу. Если выбранный PCB является последним, то соответствующий бит в SCH$GL_COMQS сбрасывается (строки 27—43).

4. Приоритет изменяется так, чтобы по истечении кванта времени процесс мог быть автоматически помещен в очередь с меньшим приоритетом (строки 44—51).

5. Адрес аппаратного блока управления процессом загружается в базовый регистр PCB. После этого с помощью команды LDPCTX («Загрузка контекста процесса») производится загрузка аппаратных регистров требуемой информацией. Наконец, команда

RET («Возврат из прерывания») обеспечивает возврат из начального приоритетного уровня прерывания 3. Если в системе нет ожидающих обработки прерываний более низкого приоритета, процессор вернется к состоянию IPL0 и продолжит выполнение команд нового процесса (строки 52—55).

Здесь обращаем внимание читателя на следующее обстоятельство. Программа перепланировщика, текст которой приведен на рис. 6.5, единственное место в МОС ВП, где применена команда LDPCTX. В противоположность ей команда SVPCTX («Сохранить контекст процесса») используется чаще. В частности, она выполняется в процедурах регистрации событий в том случае, когда текущий процесс должен быть помещен в очередь ожидания. После этого происходит обращение к программе перепланирования (в точку SCH$CHED) для запуска нового процесса.

6.3. УПРАВЛЕНИЕ ПАМЯТЬЮ. СВОПИНГ

В предыдущей главе были описаны аппаратные средства управления памятью СМ1700. Архитектура этой ЭВМ обеспечивает реализацию целого ряда возможностей такого управления и среди них:

использование адресного пространства для логически линейных программ;

защита памяти;

разделенное использование программ операционной системы и данных;

перехват обращений к нерезидентным страницам.

Однако рассмотренные основные возможности аппаратных средств не обеспечивают решения всех проблем, связанных с управлением памятью. Так, они не предусматривают загрузки в память или удаления из нее отдельных страниц программы, что приводит к необходимости программной реализации соответствующих процедур. Создание подобного программного механизма потребовало учета специфических особенностей вычислительной системы и обоснованного выбора алгоритмов управления памятью.

Виртуальная память и обмен страниц. Рассматривая вопросы организации, проектирования и использования вычислительных систем с виртуальной памятью, мы не должны ни на минуту забывать о том, что они предоставляют пользователю возможность оперировать большим адресным пространством, но за это приходится расплачиваться потерей производительности. Иными словами, виртуальная память не решает извечной программистской проблемы «память или время». Во-первых, при использовании такой памяти возникают накладные расходы (и прежде всего процессорного времени), связанные с ведением страничных таблиц. За счет применения некоторых вспомогатель-

Рис. 6.6. Зависимость числа отказов по обращению к странице от отношения объемов виртуальной и физической памяти программы [5]

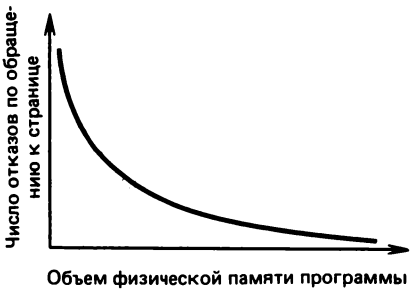


Рис. 6.7. Зависимость числа отказов по обращению к странице от объема физической памяти программы [5]

ных аппаратных средств эти расходы можно весьма существенно уменьшить. Какие для этого нужны средства и как они функционируют, мы и рассмотрим.

Во-вторых, что более существенно, весьма высокой оказывается стоимость *обращения к нерезидентной* (т. е. отсутствующей в памяти) *странице*. В ряде случаев такое обращение может потребовать как записи на диск, так и чтения с него. Эти действия занимают много времени и связаны с большим объемом работы центрального процессора, обусловленным выполнением большого числа операций самой операционной системой. Следовательно, задачу повышения эффективности управления виртуальной памятью можно рассматривать как одну из самых важных с точки зрения увеличения производительности оснащенных ею вычислительных систем. Решение данной задачи возможно, в частности, за счет назначения каждому процессу такого объема оперативной памяти, который обеспечивает минимизацию числа обменов страниц с дисками. Очевидно, что при этом должно учитываться и количество параллельных процессов, осуществляющих подобные обмены.

Число *отказов по обращению к странице* памяти зависит от значений целого ряда параметров: интенсивности запросов к различным участкам адресного пространства, физического разделения памяти между процессами, стратегии распределения страниц, реализуемой операционной системой, и др. Важнейшим из них с точки зрения количества неудовлетворенных запросов к памяти является ее физический объем, приданный процессу. Такую зависимость демонстрирует рис. 6.6, на котором приведен типовой график числа отказов по обращению к странице как функции отношения объемов виртуальной и физической памяти процесса. В пределе, когда такое отношение равно 1, все обращения удачны, поскольку все данные и коды расположены в оперативной

памяти. По мере роста значения этой величины увеличивается и число отказов по обращению к страницам. Типовая зависимость числа отказов от объема выделенной процессу физической памяти приведена на рис. 6.7. Из этого графика следует, что, начиная с некоторого момента, увеличение объема физической памяти, принадлежащей процессу, не обеспечивает существенного снижения числа неудачных обращений к страницам. Указанный момент наступает, как правило, задолго до того, как процесс становится полностью резидентным. Следовательно, необходимо, чтобы операционная система по возможности предупреждала перерасход памяти процессом, поскольку это не дает никаких выгод пользователю, ущемляя интересы других. По мере функционирования процесса происходит обращение к различным его адресам. В том случае, когда требуемой страницы нет в физической памяти, происходит ее считывание с диска. В результате в некоторый момент процесс может занять весь выделенный ему объем памяти. В дальнейшем работает следующий механизм. При обращении к адресам, принадлежащим странице, отсутствующей в памяти, управление передается операционной системе, которая освобождает участок физической памяти и записывает в него затребованную страницу. Стратегия, определяющая правила подобных замещений, называется *алгоритмом смены страниц.*

Описанная в [5] оптимальная стратегия замещения использует полную информацию об удачных и неудачных обращениях программы к страницам памяти. Причем для удаления из памяти выбираются те страницы, к которым в течение достаточно длительного времени не предполагаются обращения. Учитывая, что обычно такая информация является недоступной, для предсказания будущего поведения программы используются сведения о ее функционировании в прошлом. Например, в схеме *последнего использования* (схема LRU) в качестве удаляемой выбирается та страница, к которой дольше всего не было обращений, поскольку предполагается, что и в достаточно близком будущем эта страница окажется ненужной. Очевидно, что алгоритмы, подобные LRU, требуют организации сбора информации о частоте использования каждой страницы. В большинстве случаев это реализуется аппаратно, для чего таблицы страниц содержат так называемый бит обращения, устанавливаемый в единицу при каждом обращении к соответствующей странице. Если с помощью программных средств этот бит периодически сбрасывать, то он может быть использован для прослеживания числа обращений.

Более эффективную интуитивную оценку поведения процесса можно получить путем использования модели, предложенной в уже цитировавшейся работе [5]. Здесь вводится понятие *рабочего множества* процесса $W(T, t)$, представляющее собой набор страниц, к которым в интервале времени $(T-t, T)$

произошло обращение. Если длительности интервалов t оказываются различными, то рабочим множеством является совокупность страниц, затребованных в течение последних t секунд, а мощность $W(T, t)$ равна числу таких страниц. Как состав рабочего множества, так и его размер могут изменяться за время функционирования процесса, в частности, при увеличении T. Соответствующим подбором значения t (размера окна) механизм управления памятью обеспечит резидентность тех фрагментов адресного пространства процесса, которые входят в его рабочее множество. При выборе оптимального размера окна должны быть учтены быстродействие процессора и возможности аппаратуры сегментации памяти.

Рассмотренная модель описывает некоторый базис, обеспечивающий сохранение в памяти страниц, необходимых процессу. Для заданного t те из них, к которым в течение всего последнего окна не было обращений, удаляются из памяти с целью сохранить постоянным рабочее множество.

Существует определенная зависимость между стоимостью, связанной с отказами при обращении к странице, и сложностью алгоритма, используемого для уменьшения их числа. Если процедура замещения достаточно эффективна в смысле уменьшения числа отказов при обращении к памяти, но для своей реализации требует чрезмерно больших затрат процессорного времени, то она не может быть признана заслуживающей внимания. В этом случае имеет смысл воспользоваться более простыми и требующими меньших затрат *алгоритмами случайного замещения*, реализующими замещение страниц случайным образом, или *алгоритмом* FIFO, обеспечивающим удаление той из них, которая находится в памяти дольше всех. При использовании механизма FIFO система следит только за очередью станиц. Новая страница добавляется каждый раз в конец очереди, а самая старая удаляется из ее начала.

Независимо от того, насколько «интеллектуальна» стратегия замещения страниц, все-таки основным фактором, влияющим на эффективность использования памяти, является размер ее физического пространства. Очевидно, что при любом алгоритме число отказов по обращению к странице будет тем большим, чем меньший объем физической памяти выделен процессу (см. рис. 6.6). Однако в ряде случаев снижением производительности, вызываемым сменой страниц, можно поступиться ради возможности организовать функционирование процессов на адресном пространстве, превышающем объем физической памяти. А именно это и обеспечивается использованием виртуальной памяти.

Управление памятью в МОС ВП. Мы уже рассматривали аппаратные средства управления памятью в СМ1700. Здесь же обсудим вопросы организации и функционирования аналогичного программного механизма, реализованного в МОС ВП. Он состоит

из двух подсистем, осуществляющих частичный или полный обмен фрагментов адресного пространства процесса с внешней памятью на дисках. Первая из них — *обработчик страниц* PAGER, представляет собой набор процедур, функционирующих в контексте процесса в режиме ядра. Основной задачей PAGER является обеспечение доступности нерезидентных страниц процесса, при обращении к которым произошел отказ. В ряде случаев для этого некоторые страницы исключаются из числа резидентных. Обработчик разделяется всеми активными процессами, т. е. он получает управление и выполняется от имени некоторого процесса каждый раз, когда в последнем происходит обращение к отсутствующей в оперативной памяти странице. Другим компонентом МОС ВП, реализующим управление памятью, является *обменник* SWAPPER, который выполняет выгрузку из памяти или загрузку в нее процесса целиком. Обменник тесно связан с планировщиком, они совместно определяют, какой из процессов должен быть удален из памяти на диск или введен обратно. По этой причине SWAPPER называется еще планировщиком памяти.

Важной характеристикой любого процесса в МОС ВП является *квота* или число страниц физической памяти, которые он может занимать. Этот объем памяти получил название «*предельный рабочий* (*резидентный*) *набор*». Для контроля за расходом страниц система ведет список рабочего набора любого процесса, в котором перечислены все используемые страницы.

Рассмотрим теперь, как осуществляется обеспечение физической памятью вновь запускаемого процесса. Для этого сначала проводится анализ файла, содержащего его образ (т. е. коды, данные и необходимые для его существования управляющие структуры), и на основании имеющейся там информации формируются соответствующие таблицы страниц PT. Выше мы уже обсудили формат их записей или элементов PTE и, в частности, говорили о разряде достоверности (разряд 31). Так вот, в начальный момент разряды достоверности всех таблиц сброшены, поскольку еще не загружена ни одна страница. Иными словами, первоначально рабочий набор процесса пуст и его список не содержит ни одного элемента.

Как только управление передается на стартовый адрес процесса, происходит отказ по обращению к странице. В результате начинает работать PAGER, который и загружает в память соответствующий элемент файла образа. Еще до того, как это произойдет, аппаратно выполняются некоторые действия, связанные с управлением памятью: счетчик команд, установленный на адрес команды, выполнение которой привело к страничному отказу, и виртуальный адрес такой команды сохраняются в стеке. Запоминание содержимого счетчика команд необходимо для нормального возобновления процесса, а адрес команды использует

PAGER для вычисления элемента PTE таблицы PT, описывающего страницу, при обращении к которой произошел отказ. Такой элемент содержит информацию, по которой определяется адрес требуемой страницы на диске. Несколько упрощено процедуру чтения содержимого страницы в оперативную память можно представить следующим образом. Обработчик PAGER получает затребованную страницу физической памяти. Далее он меняет содержимое сответствующего PTE: устанавливает бит достоверности и заполняет поле номера страницы так, чтобы адресовать выделенную физическую память. Понятно, что если считана первая страница процесса, то она окажется первой и в резидентном наборе (значит, ей будет соответствовать первый элемент его списка). Затем информация (коды или данные процесса) вводится с диска и PAGER переводит процесс на пользовательский уровень, что обеспечивает рестарт команды, на которой произошел отказ по обращению к странице.

По мере выполнения процесса происходит все большее число страничных отказов. В каждом таком случае PAGER считывает нужную страницу с диска и формирует для ее отображения признак достоверности в PT. Если программа была написана с использованием большого объема виртуальной памяти, то возможно, что в некоторый момент рабочий набор соответствующего процесса достигнет предельного размера. С этого момента объем занимаемой физической памяти остается неизменным и все последующие считывания страниц, для которых получен отказ по обращению, будут выполняться за счет освобождения занимаемых участков физической памяти, т. е. за счет выгрузки на диск других страниц процесса.

Теперь о собственно *алгоритме замещения страниц, применяемом в МОС ВП.* Его основные особенности заключаются в следующем. Во-первых, выбор очередной страницы для удаления из памяти осуществляется для каждого отдельного процесса автономно, т. е. если им занята вся допустимая физическая память (достигнуто значение предела рабочего набора), то ни одна его страница не может быть использована каким-либо другим процессом. В противоположность этому во многих системах с виртуальной памятью распределение памяти в явном виде не связано с процессами и при выборе очередной страницы для выгрузки на диск рассматривается все физическое пространство. Во-вторых, в МОС ВП принят тривиальный циклический алгоритм замещения (первым пришел — первым обслужен). Применительно к управлению памятью это означает, что при возникновении страничного отказа в некотором процессе, из его резидентного набора (если израсходована вся квота) удаляется страница, загруженная в память раньше всех остальных.

Хотя данный метод и не является оптимальным, но, как будет показано в дальнейшем, плата за ошибочное удаление,

т. е. за удаление из памяти той страницы, к которой очень вероятно обращение, весьма невысока. К тому же реализация соответствующих процедур требует существенно меньших затрат, чем, скажем, реализация механизма удаления страницы, к которой дольше всего не было обращения. В последнем случае необходимы выполняемые аппаратно фиксация времени последнего обращения, просмотр и модификация таблиц РТ. (Разумеется, все это можно сделать и программно, но только за счет значительных временных потерь.)

В дополнение к описанному алгоритму для увеличения эффективности использования виртуальной памяти в системе МОС ВП заводятся и обслуживаются *список свободных* и *список модифицированных страниц*. Первый из них служит источником страниц физической памяти для всех существующих в системе процессов: как только любому из них потребуется загрузка в память новой страницы (т. е. произошел страничный отказ), из этого списка выбирается очередной элемент и соответствующая страница закрепляется за процессом. Список модифицированных страниц необходим для временного хранения содержимого страниц, уже не принадлежащих рабочим наборам процессов, и для организации более эффективной выгрузки страниц на диски.

Рассмотрим, как оба списка позволяют повысить эффективность использования физической памяти СМ1700 и производительность системы в целом.

Пусть некоторый процесс обратился к виртуальному адресу, отображенному в странице, не входящей в его рабочий набор. Если при этом исчерпана квота процесса (достигнут предел числа использованных страниц), то система управления памятью удалит из набора страницу, загруженную раньше других, и введет новую. Отметим, что такая новая страница выделяется с учетом содержимого обоих списков: если в одном из них найдена ранее исключенная из рабочего набора процесса страница с затребованным виртуальным адресом, то именно она и будет вновь введена в набор. В противном случае будет выделена та, которая соответствует первому элементу списка свободных страниц. В то же время очередная выведенная из рабочего набора страница некоторое время будет храниться в каком-либо списке.

Можно увидеть некоторую аналогию в использовании кэш-памяти и обсуждаемых списков страниц. И тот и другой механизм за счет временного сохранения данных в области, непосредственно доступной системе, позволяет повысить ее производительность. Очевидно, что так же, как и в случае кэш-памяти, эффект применения списков в значительной степени зависит от их размеров и активности процессов. Описанный подход имеет и свои недостатки, среди которых в первую очередь отметим возможность быстрого переполнения списков при обслуживании одного или нескольких процессов, характеризующихся большой частотой обмена страниц. (Думается, что теперь читатель легко

может сам дать объяснение, почему МОС ВП всегда пытается сохранить незанятыми хотя бы минимальное число физических страниц.)

Заполнение списков выполняется следующим образом. Когда страница удаляется из рабочего набора, анализируется состояние бита модификации ее PTE. Его нулевое содержимое означает, что в данный момент на диске имеется точная копия страницы и нет смысла копировать ее вновь. Такая страница размещается в конце списка свободных. В том случае, когда бит модификации установлен в 1, содержимое страницы должно быть переписано на диск, а она сама зафиксирована как последняя в списке модифицированных.

Здесь сто́ит остановиться еще на одной особенности использования списка модифицированных страниц в МОС ВП. Дело в том, что помимо выполнения функций своеобразной кэш-памяти сохранение модифицированных страниц в списке позволяет организовать их *кластерную (групповую) запись* на диск. Это, во-первых, обеспечивает минимизацию системных потерь за счет существенного уменьшения числа операций ввода-вывода, а, во-вторых, ожидание страницей вывода (накопления в списке числа страниц, составляющих кластер) как раз и обеспечивает возможность ее возврата в рабочий набор процесса без вывода на диск. Запись такой страницы не производится также при завершении процесса.

В дополнение можно сказать, что, поскольку за один раз производится одновременный вывод большого числа страниц, система может попытаться произвести запись виртуально смежных страниц процесса в смежные блоки магнитного диска. Аналогично возможно *кластерное считывание*. Когда процесс обращается к одной странице, система может принять решение набрать их сразу несколько, разумеется, если соответствующие записи на диске смежны, т. е. образуют кластер.

Максимальная выгода от кластерного чтения страниц, как правило, достигается в начале выполнения нового процесса. Действительно, наиболее часто страничные отказы возникают, когда начинает формироваться рабочий набор. Поэтому при возникновении первого страничного отказа МОС ВП сразу считывает в память большое число страниц, что существенно снижает число страничных отказов, которые могут произойти в дальнейшем, и уменьшает затрачиваемое на обмен время.

Использование единой методики как для уменьшения интенсивности страничного обмена, так и для оптимального разделения времени процессора позволяет динамически изменять предельные размеры рабочих наборов. В МОС ВП имеются механизмы для выравнивания частоты страничных отказов у всех выполняемых процессов. Каждый раз по истечении кванта времени система производит анализ числа отказов,

зарегистрированных за последний временной интервал. В зависимости от полученного значения предельная величина рабочего набора может как увеличиваться, так и уменьшаться. Для процессов, имеющих высокую частоту страничных отказов, квота физической памяти увеличивается, для других (вообще не получавших отказов по обращениям к страницам или получивших их весьма мало) — уменьшается. В результате у процессов, которые не нуждаются в том количестве физической памяти, которое им было выделено изначально, излишки изымаются и передаются другим, которые действительно нуждаются в дополнительных страницах.

Свопинг процессов. Помимо управления страничной организацией система МОС ВП реализует обмен между оперативной памятью и дисками на уровне целых процессов. Как мы уже говорили, такой обмен производится программой SWAPPER, которая сама по себе процесс и никогда не выгружается. Поскольку SWAPPER оперирует несколько иными понятиями, отличными от тех, к которым мы уже привыкли, целесообразно кратко на них остановиться.

Назовем *балансным набором* все множество страниц физической памяти, которое принадлежит всем активным резидентным процессам. В этом случае рабочий набор процесса представляет собой часть или *раздел* балансного. Именно такой раздел и является единицей, которой оперирует диспетчер памяти при организации обменов между балансным набором и пространством дисковой памяти. За счет этого система предоставляет физическую память одним процессам, временно прерывая выполнение других. В целях минимизации потерь при осуществлении чтения-записи раздела балансного набора все они формируются на диске в виде кластеров.

В некоторых системах, поддерживающих виртуальную память, реализован ускоренный возврат выгруженного процесса, достигаемый за счет ввода в память некоторой части его рабочего набора. В МОС ВП процесс продолжает функционирование только после того, как будет введен весь соответствующий раздел балансного набора. Такой подход позволяет уменьшить число последующих страничных отказов и число операций ввода-вывода, поскольку кластерная организация позволяет ввести страницы в память или вывести их из нее с минимальным числом обращений к диску.

После того, как процесс вновь размещается в памяти, проходит по меньшей мере один квант до момента, когда он может быть загружен. Алгоритм, с помощью которого определяется, какой из процессов должен быть удален на диск, весьма прост. Диспетчер памяти SWAPPER анализирует очереди выполнимых нерезидентных процессов, чтобы определить подлежащий загрузке в память процесс, имеющий наибольший приоритет. Выбрав

таковой, SWAPPER осуществляет поиск фрагмента свободной памяти с числом страниц, обеспечивающим размещение рабочего набора выбранного процесса. Требуемая память может быть получена из следующих источников:

из списка свободных страниц;

за счет освобождения модифицированных страниц путем записи их содержимого на диск;

за счет страниц, освобождающихся после выгрузки на диск активного резидентного процесса, имеющего меньший или равный приоритет.

В последнем случае, чтобы освободить память для одного большого нерезидентного процесса, иногда необходимо выгрузить на диск два или более меньших по размерам резидентных.

На этом мы закончим рассмотрение вопросов диспетчеризации процессов в МОС ВП и разделения между ними физической памяти ЭВМ. Перейдем теперь к обсуждению очень важной для любой операционной системы проблемы — организации ввода-вывода.

6.4. УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ

Процедуры, реализующие ввод и вывод, оказываются порой наиболее сложными программами операционной системы. Частично это обусловлено тем, что внешние устройства, как правило, асинхронны, а их одновременная параллельная работа должна определенным образом синхронизироваться. Кроме того, многие из них требуют обслуживания в реальном времени. Перечисленные факторы и целый ряд других, не менее важных, послужили причиной того, что при создании любой операционной системы проектирование подсистемы ввода-вывода требует самой тщательной проработки. И здесь МОС ВП не составила исключение.

По решаемым задачам управление устройствами существенно отличается от рассмотренного нами управления процессором и памятью, процедуры управления которыми направлены на повышение эффективности использования процессора и памяти, и выполняется по большей части невидимо для пользователя. В отличие от этого с операциями ввода-вывода дело обстоит совсем иначе, хотя бы потому, что обмен с устройствами инициируется пользовательскими процессами. Выполнение запросов этих процессов осуществляется соответствующими процедурами ядра операционной системы. Собственно запрос на ввод-вывод представляет собой вызов некоторой процедуры. В том или ином виде такие вызовы программируются в любой программе, не исключая и написанных на языках высокого уровня. В последнем случае программист пишет операторы типа READ или WRITE. При трансляции или компиляции программы вместо них генерируются обращения

к подпрограммам исполнительной системы, написанным на языке уровня ассемблера и содержащим вызовы соответствующих процедур ОС.

Кроме собственно инициализации обращения к процедуре ядра вызов, связанный с запросом на ввод-вывод, служит источником информации, необходимой для выполнения обмена данными с устройством: тип затребованной операции (чтение, запись, управление и т. д.), адрес и длина пользовательского буфера и некоторые другие параметры. Такая информация (в МОС ВП она называется аргументами ввода-вывода) после некоторой обработки в соответствующей процедуре ядра передается в специальный процесс-драйвер устройства. Именно этот компонент системы осуществляет собственно обмен данными с внешним устройством. В большинстве современных операционных систем, в том числе и в МОС ВП, процессы могут инициализировать *синхронный* либо *асинхронный ввод-вывод*. Напомним читателю, что при запуске операции первого типа процесс переводится в состояние ожидания окончания обмена. При асинхронном вводе-выводе выполнение процесса продолжается параллельно с передачей приемом информации.

Обратим внимание на еще одну особенность реализации операций ввода-вывода в вычислительных системах, не имеющих аппаратно реализованных каналов (такие каналы имеются, например, в ЕС ЭВМ). Речь идет о том, что при подключении к единой магистрали ЭВМ большого числа разнотипных и разноскоростных периферийных устройств возникает проблема координации работы обслуживающих их драйверов. Исходя из этого система ввода-вывода МОС ВП должна включать механизм диспетчеризации драйверов и слежения за доступностью необходимых для обмена с оборудованием системных ресурсов.

Итак, рассмотрим, как в МОС ВП решаются задачи, связанные с организацией ввода-вывода. Для этого проследим прохождение запроса на ввод-вывод от пользовательского процесса до выхода на конкретное периферийное устройство. Учитывая сложность системы управления данными и ограниченный объем настоящей книги, основное внимание в дальнейшем будет уделяться выполнению операций обмена процедурами ядра.

Система ввода-вывода МОС ВП. Как видно из рис. 6.8, организация ввода-вывода в МОС ВП может быть представлена несколькими иерархическими уровнями, каждый из которых взаимодействует со смежными через соответствующие интерфейсы. Самый верхний, т. е. наиболее общий уровень архитектуры системы управления вводом-выводом, реализован как система RMS, выполняющая обработку поименованных объектов типа файл, запись, блок, поле и др. Нижним программным уровнем в рассматриваемой архитектурной модели является уровень, представляющий процедуры ядра, ответственные за реализацию

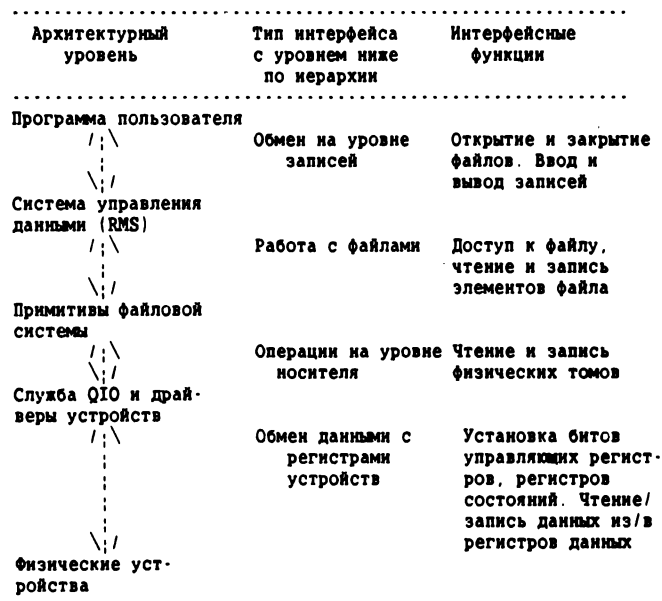


Рис. 6.8. Архитектура системы ввода-вывода МОС ВП

ввода-вывода (служба QIO), и драйверы устройств, непосредственно связанные с оборудованием.

Как уже говорилось, в МОС ВП допустимы как асинхронные, так и синхронные операции ввода-вывода. Таким образом, при написании программы есть возможность выбрать один из трех подходов к организации обмена данными с внешним устройством: запустить операцию, указать адрес подпрограммы обработки асинхронного прерывания по завершению обмена и продолжить выполнение; запустить операцию с указанием флага, который должен быть установлен при окончании обмена, продолжить выполнение с периодической проверкой установки флага; запустить операцию с указанием флага, однако далее перейти в состояние ожидания его установки, т. е. завершения операции. Мы уделили так много внимания рассмотрению возможных в МОС ВП режимов ввода-вывода с тем, чтобы читатель, знакомый с ОС РВ [4], мог понять сходство подсистем управления вводом-выводом, реализованных в обеих операционных системах.

На этом аналогия между системами ввода-вывода ОС РВ и МОС ВП не заканчивается. Обе они функционируют на основе весьма близких принципов и пользуются похожими таблицами или *структурами базовых данных*, описывающими физические устройства. (К слову сказать, такие таблицы описывают и виртуальные устройства, но во избежание внесения неясностей

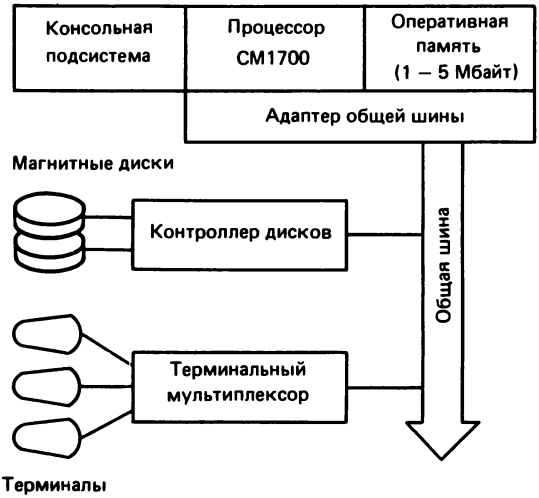


Рис. 6.9. Пример конфигурации технических средств СМ1700

остановимся пока только на реальных.) Поскольку структуры базовых данных играют очень важную роль в организации обмена информацией с устройствами, представляется целесообразным рассмотреть их назначение и связи между ними.

На рис. 6.9 изображена конфигурация СМ1700, а на рис. 6.10 приведен конкретный пример таблиц, описывающих данную конфигурацию, а также представлены их взаимосвязи. Кроме того, на рис. 6.11 даны также связи между пользователем системы ввода-вывода, структурами базовых данных и драйвером устройства. Итак, при выполнении операции ввода-вывода привлекаются следующие элементы ядра МОС ВП.

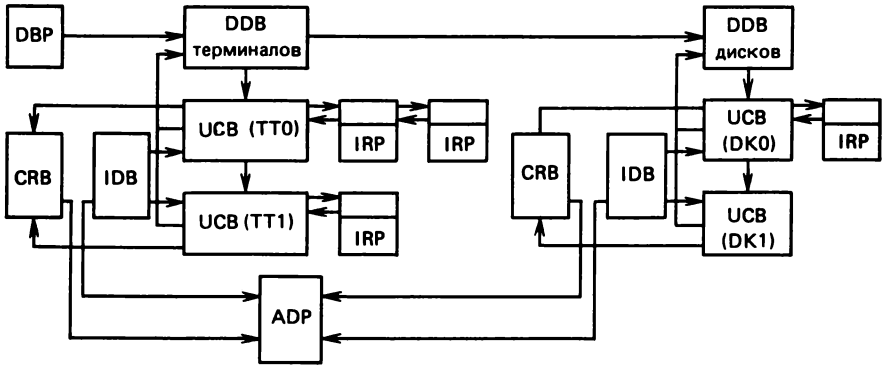


Рис. 6.10. Структура базовых данных системы ввода-вывода

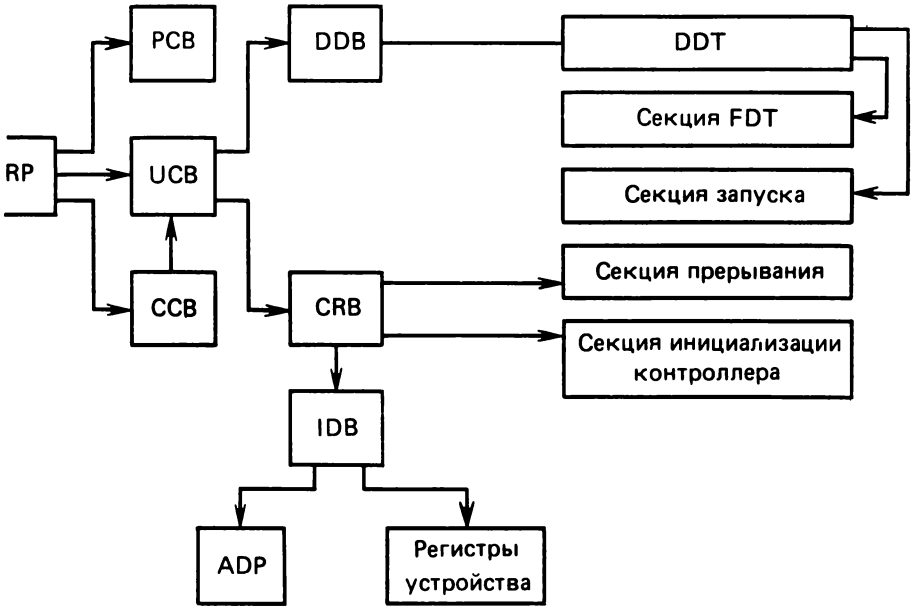


Рис. 6.11. Взаимосвязи структур базовых данных и компонентов драйвера

Пакет запроса ввода-вывода IRP описывает пользовательский запрос на операции обмена с внешним устройством. В частности, он содержит ссылку на PCB затребовавшего обращения к устройству процесса, описание запрошенной операции, адрес и длину пользовательского буфера и т. д.

Программный *блок управления процессом* PCB, как нам уже известно, описывает затребовавший операцию процесс.

Блок управления каналом CCB описывает логическую связь между процессом и блоком UCB назначенного процессу устройства. Когда при выполнении процесса им вызывается служба назначения канала ввода-вывода, осуществляется запись адреса соответствующего UCB в CCB.

Блок управления устройством UCB описывает характеристики и текущее состояние конкретного устройства. Этот блок используется также для запоминания данных (необходимых драйверу и той части его контекста, которая зависит от устройства) во время выполнения операции.

Блок данных устройства DDB содержит информацию, общую для всех устройств одного и того же типа, подсоединенных к отдельному контроллеру. Блок DDB содержит имя типа устройства, имя и адрес его драйвера. Кроме того, этот блок содержит и адрес UCB первого устройства, подсоединенного к соответствующему контроллеру.

Блок запроса канала CRB строится системой для каждого имеющегося в конфигурации СМ1700 контроллера и описывает его текущее состояние. В CRB запоминается также список устройств, ожидающих освобождения канала данных контроллера, и код, позволяющий идентифицировать подпрограмму обработки прерывания от устройства.

Диспетчерский блок прерывания IDB создается МОС ВП для каждого контроллера. В некотором смысле IDB можно рассматривать как логическое продолжение CRB, поскольку в диспетчерском блоке прерывания перечислены все устройства, связанные с соответствующим контроллером, и указывается UCB того из них, которое в текущий момент обслуживается. Кроме того, IDB ссылается на регистры контроллера и адаптер шины.

Блок управления адаптером ADP описывает характеристики и текущее состояние адаптера общей шины СМ1700. Он содержит также ссылки на очереди и битовую таблицу распределения, необходимые для распределения ресурсов адаптера. Ядро МОС ВП включает подпрограммы, которые могут использоваться драйверами для связи с адаптером общей шины.

Как видно из рис. 6.10, все DDB связаны в цепочку и адрес первого из них хранится в специальной ячейке DBP, что позволяет службе назначения канала быстро найти адрес UCB требуемого устройства. Для того чтобы завершить рассмотрение значения структур базовых данных в организации ввода-вывода, обратимся еще раз к рис. 6.11. На нем приведена связь таблиц драйвера с управляющими блоками; DDB с таблицей входов в драйвер или, иначе, таблицей диспетчеризации драйвера DDT, блока CRB с секциями обслуживания прерываний и инициализации контроллера. На первую из них МОС ВП передает управление при возникновении прерывания от устройства, а на вторую — при начальной загрузке системы либо во время восстановления после нарушения питания. И, наконец, таблица DDT служит основой для выбора необходимой подпрограммы предобработки или FDT-подпрограммы (она выбирается по специальной таблице решений функций драйвера), а также содержит ссылки на секцию запуска ввода-вывода. В FDT-подпрограмме осуществляется проверка допустимости значений, специфичных для устройства параметров запроса ввода-вывода, размещение буферов, фиксация требуемых страниц в памяти и выполняется ряд других функций, предшествующих началу обмена данными с устройством. А собственно обмен начинается как результат работы секции запуска.

Элементы МОС ВП, ответственные за обмен информацией с периферийными устройствами, объединены в три службы. О первой из них — *службе обслуживания очередей ввода-вывода* или *службе* QIO, мы уже неоднократно упоминали. Обращение к ней производится каждый раз, когда система получает запрос на выполнение операции. В сущности служба QIO представляет

собой процедуру ядра, которая непосредственно вызывается пользовательским процессом. Хотя QIO и выполняется в режиме ядра, тем не менее при работе она использует контекст процесса пользователя и имеет доступ к его адресному пространству. Основные ее функции заключаются в проверке допустимости передаваемых пользователем аргументов, построения пакета IRP и постановке его в очередь, заголовок которой хранится в блоке UCB соответствующего устройства (см. рис. 6.10). Иногда эти общие для всех устройств проверки и действия называются *предобработкой, не зависимой от устройства.* Поведение запросившего операцию обмена процесса в дальнейшем определяется, как мы уже говорили, типом запроса: процесс либо продолжает обработку (асинхронный ввод-вывод), либо переводится в состояние ожидания завершения обмена (синхронный ввод-вывод).

И о второй службе системы ввода-вывода МОС ВП мы уже упоминали — о *драйвере устройства.* Он представляет собой набор секций подпрограмм и таблиц. Рассмотрим кратко основные из них.

Секция предобработки включает подпрограммы, осуществляющие специфические проверки и действия, необходимые конкретному устройству для запуска операции, а также размещение буферов и фиксацию страниц в памяти.

Секция запуска, или стартовая, где производится активизация устройства.

Секция прерываний, ответственная за обработку прерываний, возникших от обслуживаемых драйвером устройств.

Секция обработки ошибок, в которой производится (разумеется, если это возможно) восстановление прекращенной по ошибке операции ввода-вывода и выполняется ряд других действий, связанных с различными исключительными ситуациями.

Секция подготовки сообщений об ошибках включает подпрограммы, обеспечивающие фиксацию содержимого регистров устройства и других данных в специальном буфере (для их последующей обработки).

Секция отмены, предотвращающая дальнейшую обработку запроса ввода-вывода.

Секция инициализации, где производится подготовка контроллера или устройства в момент начальной загрузки системы либо ее восстановления после сбоя питания.

Прологовая таблица, которая необходима для загрузки драйвера. Она содержит описания драйвера и типа сответствующего устройства.

Диспетчерская таблица драйвера DDT, в которой перечислены все адреса входов в секции, размеры буферов для диагностики и сообщений об ошибках устройства.

Таблица решений функций FDT необходима для выбора подпрограммы предобработки в зависимости от кода затребованной функции ввода-вывода.

С точки зрения процесса — основной программной структуры, которой оперирует система МОС ВП, драйвер представляет собой так называемый *форк-процесс*, создаваемый динамически и имеющий укороченный контекст. Последнее означает, что драйвер не может выполнять операции в пределах пользовательского контекста, он оперирует только пространством системных адресов. Состояние форк-процесса драйвера полностью определяется его программным счетчиком, содержимым трех универсальных регистров и содержимым блока управления устройством.

Драйвер использует центральный процессор на высоком приоритетном уровне. Следовательно, его работа не может быть прервана планировщиком и продолжается либо до полного завершения обслуживания операции ввода-вывода и перехода драйвера в ожидание очередного запроса, либо до его прерывания другим форк-процессом, работающим на более высоком приоритетном уровне. Пока хотя бы один драйвер целиком не завершил свою работу, не может выполняться ни один пользовательский процесс. При прерывании форк-процесса для сохранения его контекста используется UCB, т. е. по отношению к драйверу этот блок выполняет ту же роль, что и PCB для пользовательского процесса.

Третьей службой системы ввода-вывода МОС ВП является набор *процедур постобработки*, служащих для выполнения всех действий, связанных с завершением операций обмена. При выдаче запроса на ввод-вывод процесс может специфицировать флаг события, который должен быть установлен при завершении операции, указать адрес подпрограммы обработки соответствующего асинхронного системного прерывания, зарезервировать место для специального блока состояния ввода-вывода, в который должен быть записан код завершения операции. Кроме того, даже если ничего этого пользователь не предусмотрел при разработке своей программы, тем не менее необходимо по крайней мере дать возможность процессу обращаться к полученным от устройства данным. Перечисленные действия не могут быть выполнены драйвером, поскольку он не имеет доступа к адресному пространству затребовавшего операцию процесса. Именно это определило необходимость включения в ядро системы процедур постобработки.

Рассмотрим теперь, как в МОС ВП организовано взаимодействие описанных служб системы ввода-вывода. Пусть пользовательский процесс запросил ввод-вывод. В несколько упрощенном виде последовательность выполняемых при этом действий представлена на рис. 6.12. Как видно из рисунка, пользовательский процесс генерирует обращение к службе QIO. Последняя получает управление в режиме ядра, но функционирует в контексте процесса пользователя. На основании результатов анализа содержимого блока запроса, содержащего всю сопутствующую запросу информацию, служба QIO определяет допустимость запроса. При положительном исходе проверки, т. е. нормальном завер-

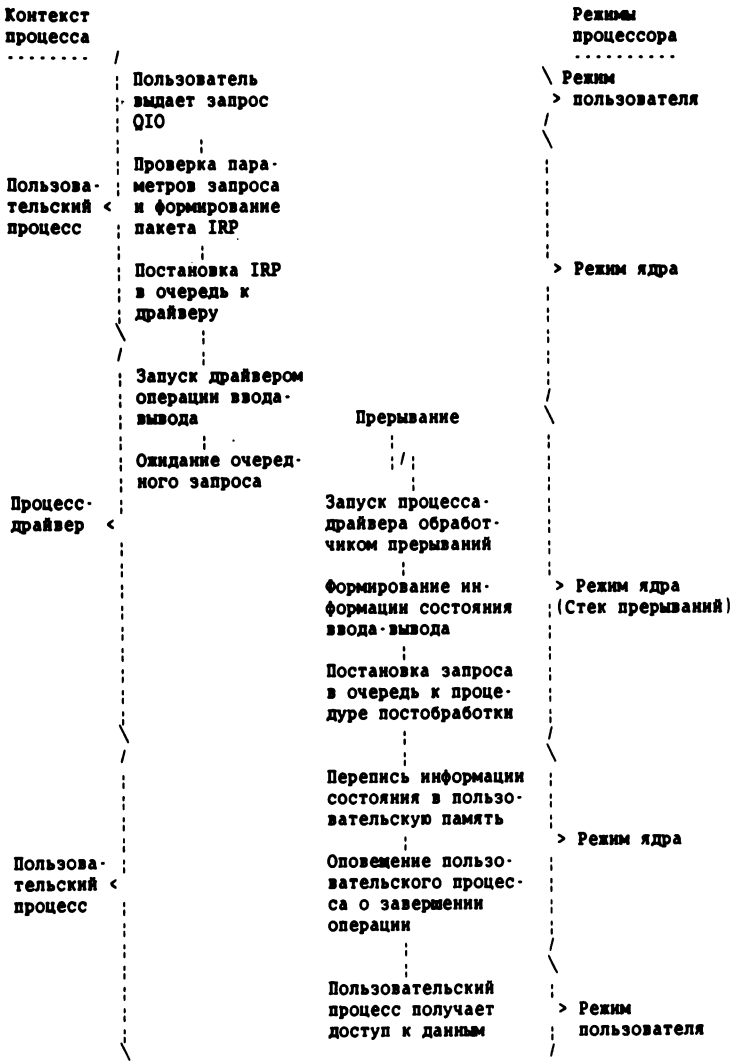


Рис. 6.12. Обработка запроса на ввод-вывод

шении предобработки, независимой от устройства, выполняется процедура предобработки, учитывающая специфику устройства. Для этого служба очередей по таблице решения функций драйвера идентифицирует требуемую FDT-подпрограмму и активизирует секцию предобработки драйвера, работающую в режиме ядра на контексте пользовательского процесса. При положительном исходе всех описанных проверок, управление вновь попадает

в службу QIO, которая формирует соответствующий пакет IRP и ставит его в очередь к драйверу устройства. Заголовок подобной очереди имеется в любом UCB.

После полного завершения предобработки служба QIO передает управление процедуре ядра, ответственной за создание форк-процессов для драйверов. В качестве содержимого программного счетчика порождаемого процесса принимается адрес секции запуска, выбираемый из диспетчерской таблицы DDT. Мы уже отмечали, что роль PCB для процессов с укороченным контекстом выполняет UCB соответствующего устройства. Каждый блок управления устройством включает специальный набор полей (подблок), называемый форк-блоком. Именно такой форк-блок (в целях совместимости с ОС РВ его можно было бы называть блоком отложенной обработки) ставится в очередь форк-процессов.

Следующий этап обработки запроса ввода-вывода осуществляется уже в контексте форк-процесса драйвера, функционирующего в режиме ядра. На этом этапе секция запуска ввода-вывода активизирует устройство путем записи содержимого в его регистры. Теперь обработка текущего запроса приостанавливается и выполнение форк-процесса откладывается до завершения операции на устройстве, т. е. до возникновения прерывания от него. Сохраняемый для возобновления работы драйвера контекст указывает на адрес внутри секции запуска как на точку возобновления форк-процесса.

Как видно из рис. 6.12, после распознавания операционной системой прерывания от устройства, на котором была запущена операция, форк-процесс возобновляет свою работу в точке входа в секцию обслуживания прерываний. На этой стадии драйвер оперирует как содержимым IRP, так и битами состояния устройства и контроллера. В том случае если прерывание вызвано окончанием физической операции, то осуществляется подготовка необходимых таблиц и общих регистров для использования занесенных в них данных в процедуре постобработки.

Как и рассмотренные выше операции по подготовке к запуску и обмену с устройством, постобработка выполняется в два приема: в контексте форк-процесса осуществляется обработка, обусловленная спецификой устройства, а в контексте пользовательского процесса — общая для всех устройств. Из рис. 6.12 видно, что на уровне форк-процесса формируется информация о завершении запрошенной пользователем операции ввода-вывода. Поскольку форк-процесс имеет укороченный контекст, далее он лишь ставит пакет запроса в очередь к процедуре постобработки (выполняемой в контексте процесса), чтобы окончательно завершить обработку запроса: записать коды завершения операции в блок состояния ввода-вывода, при необходимости скопировать принятые данные в буфер пользователя и т. д.

Переключение на контекст пользовательского процесса производится с помощью *асинхронного системного прерывания*, или

AST-*прерывания*. Его результатом будет запуск внутри процесса и асинхронно по отношению к нему системной AST-подпрограммы. Система МОС ВП допускает кроме этих специфицированных процедурой постобработки еще и AST-подпрограммы, заданные пользователем. Для любого процесса система создает и обслуживает очередь управляющих блоков AST (AST-очередь) с заголовком, хранящимся в программном PCB. Каждый элемент такой очереди описывает некоторое системное асинхронное прерывание и содержит адрес входной точки подпрограммы, на которую передается управление в момент его возникновения. Если исходный процесс в этот момент выполняется, то происходит его прерывание. После завершения AST-подпрограммы процесс продолжает работать.

Итак, для завершения обслуживания запроса на ввод-вывод процедура постобработки формирует и ставит в контексте драйвера в AST-очередь управляющий блок, в котором задан адрес одной из ее подпрограмм, выполняемых на контексте процесса в режиме ядра. В результате эта подпрограмма является с точки зрения операционной системы обычной подпрограммой обработки асинхронных прерываний, активизируется стандартным образом и выполняет все действия, необходимые для окончания обработки запроса на ввод-вывод. В частности, если процесс затребовал запуска собственной AST-подпрограммы после завершения операции обмена, необходимым образом будет модифицирована AST-очередь.

Ранее мы говорили, что при выполнении запросов на ввод-вывод в МОС ВП используются различные приоритетные уровни прерываний. Рассмотрим теперь, как это реализуется и что дает.

Приведенное выше описание последовательности действий при обработке запроса на обмен с внешним устройством, поступившего от пользовательского процесса, относится к единичной операции ввода-вывода. Легко представить себе, что в такой большой многопользовательской системе, как МОС ВП, несколько требований на обмен данными могут находиться на разных стадиях обработки. Следовательно, система ввода-вывода параллельно решает несколько задач, обеспечивая одновременное управление всеми такими операциями. Более того, при этом осуществляется попытка оптимизировать суммарную производительность и одновременно обеспечить функционирование в реальном времени. Повышение производительности возможно только за счет максимальной загрузки устройств, и именно поэтому в МОС ВП едва заканчивается обмен данными с некоторым устройством, система тут же запускает на нем новую операцию из очереди к соответствующему UCB.

Однако все это не может обеспечить реакцию, необходимую для работы в реальном времени. Для удовлетворения требований такого режима в системе ввода-вывода МОС ВП используется другой механизм: приоритетность уровней

программных прерываний. Иначе говоря, при обработке запросов на ввод-вывод различные функции выполняются на разных приоритетных уровнях. В результате вместо полного обслуживания каждого из последовательно поступающих запросов для всех сразу выполняются функции, наиболее критичные по времени, затем менее критичные и так далее. Говоря «для всех», мы имеем в виду те запросы, которые в рассматриваемый момент уже обслужены процедурой QIO.

В МОС ВП выделяются четыре уровня работ по выполнению операций ввода-вывода. В табл. 6.2 показано, какие функции реализуются на каждом уровне. Еще раз подчеркнем, что система ввода-вывода МОС ВП структурирована таким образом, что должны завершиться все операции на более высоком уровне, прежде чем управление будет передано на низший.

При наивысшем уровне приоритета осуществляется обработка прерываний от устройств. В табл. 6.2 (*1*) показано, что диспетчер прерываний идентифицирует устройство, от которого поступил сигнал прерывания, и инициализирует работу секции обслуживания прерываний соответствующего драйвера. Требования, предъявляемые МОС ВП к процессу обработки прерываний, состоят в том, чтобы на него тратилось как можно меньше времени, поскольку в противном случае другие аппаратные прерывания могут оказаться заблокированными. Поэтому обслуживание прерываний осуществляется в два приема: первичная обработка, как правило, заключается в копировании содержимого регистров контроля и состояния в UCB и выполняется на приоритетном уровне прерываний устройств. Вторичная обработка реализуется в форк-процессе и, следовательно, на уровне приоритета форк-IPL. Для такого понижения приоритета драйвер использует специальную системную процедуру, обращение к которой приводит к запоминанию контекста драйвера в форк-блоке UCB и постановке этого блока в очередь отложенных форк-процессов. Далее, после завершения обслуживания всех возникших аппаратных прерываний МОС ВП последовательно возобновляет отложенную обработку операций ввода-вывода на уровне форк-процессов. Эти действия отражены в табл. 6.2 (*2*).

Как только работа драйвера возобновляется, он освобождает занимаемые ресурсы, в которых больше не нуждается, в частности тракты передачи данных контроллера. Это позволяет повысить пропускную способность системы в целом. Аналогичные цели преследует синхронизация использования структур базовых данных, выполняемая на уровне отложенной обработки прерываний. Действительно, поскольку МОС ВП принудительно прерывает функционирование драйвера и возобновляет его после некоторого времени бездействия, то за счет последовательных рестартов различных форк-процессов драйверов из очереди отложенных доступ к разделяемым структурам данных обеспечивается лишь

Таблица 6.2. **Использование разного приоритета прерываний при организации ввода-вывода**

Номер последовательности событий	Причина инициализации	Приоритетный уровень	Последовательность реализуемых функций
		Наивысший приоритет	
1	Прерывание от устройства	IPL устройства	Адресация драйвера Получение информации состояния Построение форк-блока Запрос на форк-прерывание Возврат в систему (REI)
2	Программное прерывание (диспетчера)	Форк-IPL	Выборка элемента форк-очереди Возврат в систему (REI), если очередь пуста Восстановление состояния драйвера Работа драйвера Завершение работы драйвера, освобождение ресурсов Размещение информации о состоянии в IRP, постановка IRP в очередь процедуры постобработки Запрос на программное прерывание для запуска процедуры постобработки Возврат в диспетчер при отсутствии элементов в очереди к драйверу Порождение нового форк-1 процесса и запуск обработки новой операции
3	Программное прерывание	IPL процедуры постобработки	Выборка очередного IRP из очереди Возврат в систему (REI), если очередь пуста Завершение обработки в системном адресном пространстве Формирование элемента AST-1 очереди Возврат на (*3*)
4	Асинхронное системное прерывание (внутри контекста процесса)	IPL AST-процедуры	Запись информации состояния в пользовательскую память Запись данных в пользовательскую память Формирование признаков окончания операции Возврат в систему (REI)
		Самый низкий приоритет	

для одного из них, а именно для того, который в рассматриваемый момент запущен из форк-очереди. Таким образом, механизм отложенной обработки, или форк-механизм, представляет собой средство принудительной последовательной

активизации драйверов, обеспечивающее в любой момент активность на форк-уровне единственного из них.

Совершенно очевидно, что обработка запроса ввода-вывода потребует дополнительного времени центрального процессора для выполнения действий в контексте процесса. Однако система вновь замедляет ее путем понижения приоритетного уровня. Это достигается за счет перехода в процедуру постобработки ядра, для запуска которой необходимая информация запоминается в блоке запроса IRP. Последний ставится в очередь к средствам постобработки, затем драйвер выдает требование на программное прерывание, которое ассоциируется системой с AST-подпрограммой, специфицированной для завершения обслуживания операции ввода-вывода.

Читателю уже должно быть очевидно, что постобработка осуществляется только после того, как запущены все текущие операции обмена и выполнена обработка существующих запросов на более высоких уровнях прерывания. Все это также демонстрирует табл. 6.2 (*3*).

6.5. СИСТЕМА RMS И СЛУЖБА QIO

Описанные в предыдущем разделе операции, реализуемые системой ввода-вывода МОС ВП, могут быть запущены косвенно, через RMS. Возможности, предоставляемые МОС ВП для работы с устройствами, представлены схематически на рис. 6.13. Как правило, RMS, которая предоставляет более простой доступ к устройствам и независимость от них, используется для работы с накопителями на магнитных дисках, магнитных лентах и с другими устройствами массовой памяти. В том случае, если необходима специальная организация взаимодействия с периферийными устройствами либо требуется осуществить обмен с нестандартными устройствами, применяется служба QIO. Из рис. 6.13 видно, что кроме уже обсужденных нами компонентов в операциях с устройствами принимает участие вспомогательный управляющий процесс ACP. Он предназначен для организации на физическом устройстве логической структуры. При этом ACP, рассчитанный на некоторый носитель, обслуживает устройства только этого типа и, запрограммированный на создание вполне определенной файловой организации, может создавать лишь такую структуру файлов. Именно поэтому система МОС ВП включает несколько типов ACP, в частности для дисков, магнитных лент, сетевого обслуживания. Кроме того, система допускает такие режимы: каждый том внешней памяти обслуживается собственным ACP (для увеличения производительности); все однотипные носители управляются единственным ACP (в целях минимизации памяти).

Возвращаясь к рис. 6.13, подчеркнем, что кроме доступа к данным через RMS или QIO возможен подход, являющийся их комбинацией. В этом случае файл открывается и закрывается

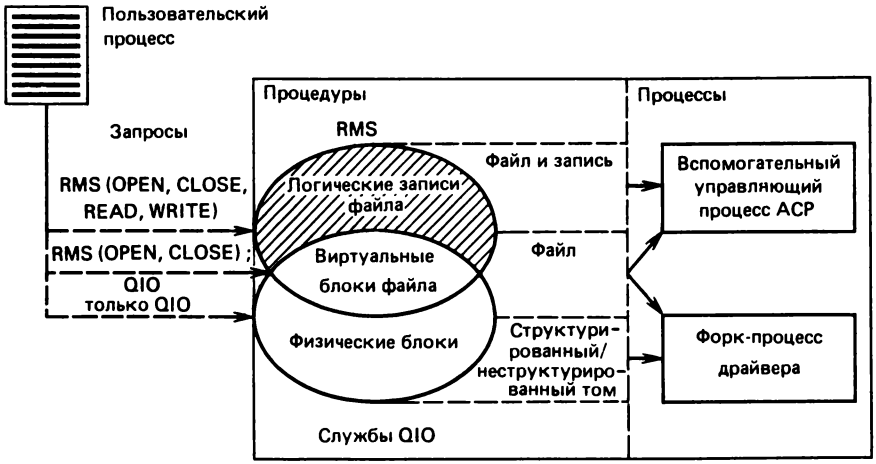


Рис. 6.13. Возможности доступа к данным в МОС ВП:
заштрихованная область – блокирование/разблокирование записей реализует RMS; незаштрихованная область – блокирование/разблокирование записей реализует пользователь

с помощью примитивов (вызовов) RMS, а все остальные операции с данными реализуются с привлечением лишь службы QIO. В табл. 6.3 представлены взаимоотношения уровней операций ввода-вывода и компонентов МОС ВП.

Система RMS, как это видно из табл. 6.3, предоставляет широкие возможности для работы с файлами с применением различных методов доступа. Она включает также средства авторизации доступа к данным, проверки корректности запросов ввода-вывода и целый ряд других. Наконец, система управления данными позволяет писать программы, инвариантные к внешним устройствам. Последнее достигается с помощью выполнения необходимых преобразований логических имен устройств и переадресации операций. В состав RMS входит большое число утилит, основные из которых приведены в табл. 6.4. Макровызовы некоторых базовых макрокоманд, составляющих интерфейс RMS с программами пользователей, даны в табл. 6.5.

Таблица 6.3. **Программные интерфейсы ввода-вывода**

Уровень операции	Интерфейс	Компоненты МОС ВП	Назначение
Запись	Вызовы RMS	RMS, ACP и драйвер	Для дисков и лент, имеющих файловую структуру, предоставляется набор методов доступа RMS, обеспечивающих ввод-вывод, который не зависит от устройства

Окончание табл. 6.3

Уровень операции	Интерфейс	Компоненты МОС ВП	Назначение
Файл	Вызовы RMS OPEN и CLOSE; вызовы QIO	RMS для OPEN и CLOSE; ACP и драйвер	На дисках и магнитных лентах с файловой структурой позволяет реализовать собственный метод доступа
Том/устройство	Вызовы QIO	Драйвер	Доступ к устройствам, не имеющим файловой структуры, или к тем, на которых реализована структура, не поддерживаемая в МОС ВП

Таблица 6.4. **Утилиты системы RMS**

Название	Функция
DEFINE	Построить файл RMS и определить его атрибуты
DISPLAY	Распечатать атрибуты одного файла или группы файлов
CONVERT	Разрешить вставку записи в индексный файл, построение последовательного файла из записей индексного или дополнение записей в последовательный файл
BACKUP	Создать копии одиночного файла или группы файлов в сжатом формате, допускающем восстановление с помощью утилиты RESTORE
RESTORE	Восстановить файл из сжатого формата, созданного утилитой BACKUP
INDEXED FILE LOAD	Заполнить индексный файл, используя записи других файлов RMS, имеющих любую организацию

Таблица 6.5. **Макровызовы базовых макрокоманд**

Уровень операции	Макровызов	Функция
Файл	$CREATE	Построить и открыть новый файл
	$OPEN	Открыть существующий файл и инициализировать его обработку
	$CLOSE	Завершить обработку файла и закрыть его
Запись	$CONNECT	Связать блок доступа к записи с требуемым файлом
	$GET	Получить запись из файла
	$PUT	Вывести новую запись в файл
	$UPDATE	Перезаписать существующую запись в файле

6.6. ИНТЕРФЕЙС СИСТЕМНОГО ОБСЛУЖИВАНИЯ

Под средствами системного обслуживания в дальнейшем понимается набор процедур ядра МОС ВП, предоставляющих пользовательским процессам разнообразные услуги. Обращение к таким процедурам производится с помощью соответствующих интерфейсных средств, которые мы рассмотрим на примере макрокоманд — одной из реализаций этого системного интерфейса.

Функции процедур системного обслуживания. Как уже упоминалось, процедуры ядра могут быть разбиты на несколько групп (см. 6.1):

информационные службы;
процедуры управления памятью;
средства управления процессами;
службы обеспечения взаимодействия процессов;
служба асинхронных системных прерываний;
процедуры ввода-вывода;
подпрограммы обработки исключительных ситуаций;
средства изменения режима.

Информационные службы МОС ВП включают процедуры, выполняющие следующие функции: выдача времени/даты; преобразование двоичного значения времени/даты в десятичный формат; преобразование двоичного значения времени/даты в печатный вид; выдача значений параметров процесса (учетной или системной информации о нем).

Время и дату по запросам пользователей предоставляет системная обслуживающая процедура SYS$GETTIM. Она оперирует специальным форматом, не всегда удобным для пользователя. Именно этим объясняется наличие в составе системных информационных служб процедур преобразования времени и даты в различное представление. С помощью SYS$GETTIM, кроме того, можно запросить информирование пользовательского процесса об истечении некоторого временного интервала (например, 30 с от момента выдачи запроса). Аналогично возможно оповещение о наступлении некоторого абсолютного времени, например 17 ч пятницы.

Весьма важной для многих прикладных задач является возможность получения по запросу почти всей информации о процессе, хранимой системой. Это, в частности, используемые умолчания, квоты, данные об использовании ресурсов и т. д. Отметим, что привилегированному процессу доступна не только его собственная информация, но и параметры других процессов.

Процедуры управления памятью. При функционировании большинство пользовательских процессов должны выполнять значительный объем функций по управлению средой. Так, они могут контролировать использование своего адресного пространства, собственное месторасположение в памяти, а также управлять размещением активных страниц. Система МОС ВП для этих целей обеспечиваст процесс следующими услугами:

занятием и освобождением страниц в пределах своего физического пространства;
блокированием и разблокированием страниц в резидентном наборе;
блокированием и разблокированием страниц в физической памяти;
блокированием и разблокированием процесса в физической памяти;
настройкой своей квоты физической памяти.

Среди системных служб, выполняющих управление памятью, следует выделить программу организации и уничтожения участков виртуального адресного пространства. Использование такой программы дает процессу возможность динамически изменять свои размеры. Например, ассемблер, которому необходимы большие таблицы, не обязательно должен резервировать место для них, поскольку за счет запроса к такой процедуре он может расширить свой размер и продолжить работу. Подобный подход значительно эффективнее статического выделения памяти, так как расширение адресного пространства происходит только в том случае, когда программа действительно в нем нуждается.

Другая особенность программ управления памятью состоит в том, что они позволяют параллельно выполняемым процессам обмениваться информацией через разделяемые страницы памяти, называемые глобальными секциями. С их помощью два процесса могут отображать некоторые участки своего виртуального пространства на одну и ту же физическую память, разделяя ее во времени. Этот механизм близок механизмам «передать по ссылке» и «получить по ссылке», реализованным в ОС РВ [4].

В той или иной степени процесс может управлять используемой им памятью. Речь идет о том, что имеющий статус привилегированности процесс может объявить себя невыгружаемым и тем самым предотвратить возможную выгрузку на диск. Данное средство полезно для всех процессов, выполнение которых критично по времени. Если запросы на решение тех или иных задач приходят в случайные моменты, то очевидно, что такие процессы должны быть резидентны в памяти независимо от того, осуществляют они обработку или находятся в состоянии ожидания поступления запросов. Альтернативой этому может служить такая организация процессов, при которой им необходимы лишь несколько страниц памяти для обслуживающей процедуры и постоянные в памяти буферы. Операционная система МОС ВП может допустить блокирование в памяти нескольких страниц такого типа процессов.

Средства управления процессами. Процесс может не только управлять своим состоянием (с применением средств управления памятью), но и при наличии статуса привилегированности влиять на другие процессы в системе. В частности, процедуры данной группы обеспечивают выполнение следующих основных функций:

создание и уничтожение процессов;

задание приоритета процесса;

перевод процесса в состояние ожидания (приостановка процесса);

рестарт приостановленного процесса;

задержка процесса.

Как следует из перечисления функций по управлению процессами, соответствующие процедуры МОС ВП позволяют одному процессу породить новый процесс (или подпроцесс), выполняемый одновременно и параллельно с породившим. Процесс-родитель в пределах выделенных ему полномочий может задать базовый приоритет порожденного процесса. Отметим, что значение приоритета может как превышать приоритет родителя, так и быть меньше.

Рассмотрим разницу между приостановкой и задержкой процесса. Самым важным их отличием является то, что повторный запуск некоторого процесса после задержки может быть инициирован любым событием из некоторого фиксированного множества. В то же время после приостановки рестарт процесса возможен только в результате обработки специального запроса, выданного параллельным процессом.

Взаимодействие процессов. Как в универсальных ОС, так и в системах реального времени часто возникает необходимость в синхронизации и обмене данными между параллельными процессами. В МОС ВП для этих целей существует несколько механизмов, работающих с различными скоростями и предоставляющих разные возможности.

1. Флаги событий. Наиболее простым аппаратом организации связей между двумя или более процессами являются флаги событий. Любой из них в программном смысле представляет собой один бит, который может быть сброшен — ожидание наступления события, либо установлен — событие произошло. Флаги могут использоваться как для синхронизации в рамках одного процесса,

так и для фиксации событий в целой системе процессов. Каждый флаг принадлежит некоторому 32-разрядному кластеру, а каждый процесс имеет доступ к четырем кластерам, номеруемым от 0 до 3. Кластеры имеют определенное назначение: кластер 0 зарезервирован для оценки поведения процессов операционной системой и только ею устанавливаются и сбрасываются соответствующие флаги; кластер 1 предназначен только для использования в рамках одного процесса; кластеры 2 и 3 являются кластерами флагов общих событий. Оба они формируются динамически по запросам к системным обслуживающим процедурам и могут использоваться одновременно несколькими процессами. Всякий раз, когда создается кластер общих событий, он получает некоторое кластерное имя. Если оно известно процессу, обладающему соответствующими привилегиями, последний может оперировать флагами такого кластера. Кластеры 2 и 3 защищаются системой МОС ВП почти так же, как файлы.

Сервисные средства МОС ВП обеспечивают следующие операции с флагами событий:

создание кластера флагов общих событий или организацию связи с ним;
сброс флага;
установку флага;
ожидание некоторого события, связанного с флагом (ожидание установки флага);
чтение кластера флагов;
ожидание дизъюнкции флагов, принадлежащих одному кластеру;
ожидание конъюнкции флагов, принадлежащих одному кластеру;
уничтожение связи с кластером;
уничтожение кластера флагов общих событий.

Ниже приведен пример синхронизации двух процессов с помощью флагов событий. Здесь в процессе 1 по макрокоманде $ASCEFC_S порождается кластер флагов общих событий с именем CLUST2. По этому же имени процесс 2 с помощью макровызова $ASCEFC_S организует связь с вновь организованным кластером. Далее оба процесса устанавливают друг для друга соответствующие флаги (флаг 65 процесс 1 для процесса 2 и флаг 66 процесс 2 для процесса 1) и переходят в ожидание сигналов синхронизации.

```
                                ;Процесс1
                                ;
$ASCEFC_S  EFN=#64,NAME=CLUST2  ;Создание кластера2 или
                                ;организация связи с ним
$SETEF_S   EFN=#65              ;Послать сигнал синхронизации
                                ;процессу2
$WAITER_S  EFN=#66              ;Перейти в ожидание сигнала
                                ;синхронизации (установки
                                ;флага 66)



                                ;Процесс2
                                ;
$ASCEFC_S  EFN=#64,NAME=CLUST2  ;Создание кластера2 или
                                ;организация связи с ним
$SETEF_S   EFN=#66              ;Послать сигнал синхронизации
                                ;процессу1
$WAITER_S  EFN=#65              ;Перейти в ожидание сигнала
                                ;синхронизации (установки
                                ;флага 65)
```

2. Почтовые ящики. Рассмотренный выше механизм флагов событий обеспечивает синхронизацию параллельно выполняемых процессов, но не

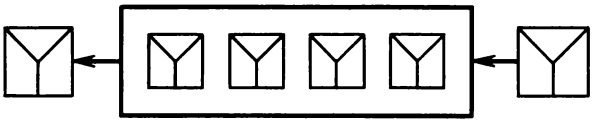


Рис. 6.14. Механизм почтового ящика

предоставляет возможностей обмена информацией между ними. Более общий вид взаимодействий процессов реализуют почтовые ящики. Каждый из них с точки зрения программы является ориентированным на записи логическим устройством. При работе с ним информация записывается и считывается так же, как из обычного файла или с физического устройства. Однако буферизация данных в этом случае происходит с применением системной динамической памяти. С помощью файловых примитивов пользователи имеют возможность открывать почтовые ящики, по командам GET и PUT читать и записывать в них информацию.

Механизм такого рода связи процессов изображен на рис. 6.14. Из этого рисунка видно, что обработка сообщений, передаваемых через почтовый ящик, возможна только в порядке их поступления. При чтении содержимого почтового ящика процесс получает либо сообщение, поступившее раньше других, либо уведомление о том, что ящик пуст. В целях обеспечения синхронизации функционирования процессов, использующих механизм почты, МОС ВП предоставляет средства информирования абонента о поступлении адресованного ему сообщения.

Почтовые ящики имеют ограниченные размеры, поэтому пользователь предварительно должен указать максимально допустимый размер сообщения и предельное их число. Одновременно возможна идентификация пользователей, которым разрешено обращение к почтовому ящику для записи сообщений. В принципе почтовые ящики могут быть защищены так же, как файлы и, следовательно, для операций над ними от процесса потребуются соответствующие привилегии.

3. Разделение памяти. Разделяемая во времени память является наиболее общим механизмом МОС ВП для организации взаимодействия процессов. Система МОС ВП допускает использование разделяемой памяти, имеющей объем, равный половине адресного пространства глобальной секции. Такая секция представляет собой поименованную область памяти, которая может быть отображена на виртуальное пространство нескольких процессов. Создание и отображение глобальной секции производится с помощью системной обслуживающей программы создания и отображения глобальной секции $CRMPSC. Для процессов, использующих разделяемую память, должны применяться специальные средства синхронизации, хотя в некоторых случаях возможна ориентация и на флаги событий или разделяемые структуры данных. Одним из основных преимуществ разделяемой памяти как средства взаимодействия процессов является то, что этот метод обеспечивает максимальную пропускную способность, поскольку не требует буферизации данных с привлечением средств операционной системы.

4. Комбинации механизмов взаимодействия процессов. Рассмотренные механизмы для организации связи между процессами могут комбинироваться с тем, чтобы события, сообщения и разделяемые данные позволяли создавать наиболее эффективные комплексы процессов, решающих сложные задачи. Рассмотрим пример функционирования системы, состоящей из трех процессов.

а) работа процесса А периодически инициируется таймером с помощью установки флага событий 1;

б) процесс А принимает данные о лабораторном эксперименте, проводит их первичную обработку, записывает в разделяемую область и уведомляет процесс В о готовности данных установкой флага событий 2;

в) процесс В анализирует данные, хранимые в разделяемой области, и определяет, что требуется изменение режимов проведения эксперимента; сообщение об этом через почтовый ящик пересылается процессу С;

г) процесс С, получив почтовое сообщение, осуществляет коррекцию параметров проводимого эксперимента так, чтобы они удовлетворяли предъявляемым требованиям; кроме того, этот процесс протоколирует проведенные действия на устройстве вывода.

Как следует из такого примера, в зависимости от конкретных требований и режимов решения задач во многих реальных случаях возникает необходимость применения некоторых комбинаций средств межпроцессорной связи. Состав комбинаций определяется в зависимости от требуемых скоростей, объемов передаваемой информации и ряда других параметров.

Обработка асинхронных прерываний. В системах реального времени, да и не только в них, процесс должен иметь возможность реагировать на некоторые события, происходящие асинхронно по отношению к нему. Такие события могут возникать в результате воздействий других процессов, вызываться периферийными устройствами или инициироваться операционной системой. Рассмотрим, как в МОС ВП реализованы процедуры обслуживания подобных событий.

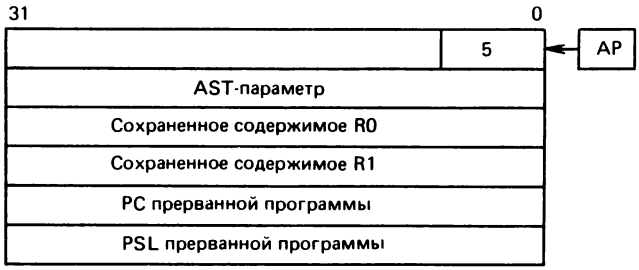
Любой пользовательский процесс, функционирующий в среде МОС ВП, имеет возможность запросить обслуживание асинхронных событий: завершение операции ввода-вывода, истечение таймерного интервала, поступление сигнала синхронизации от другого процесса и др. При выдаче такого запроса указывается адрес подпрограммы обработки асинхронного прерывания (AST-подпрограммы). Последняя представляет собой процедуру, локализованную в контексте процесса пользователя и вызываемую всякий раз, когда наступает специфицированное асинхронное событие (в МОС ВП оно оформляется как асинхронное прерывание, называемое AST-прерыванием). Если соответствующий процесс в момент возникновения AST-прерывания активен, то происходит его приостановка и управление передается на точку входа AST-подпрограммы. После завершения AST-обработки активность процесса возобновляется. Иными словами, AST-подпрограмма выполняется как параллельный AST-процесс над контекстом основного процесса. Более того, AST-процесс может служить причиной изменения состояния основного процесса: например, из-за установки некоторого флага событий отменяется состояние ожидания.

Поскольку, с одной стороны, может быть задано несколько AST-процедур, соответствующих разным AST-прерываниям, а с другой — возможно наступление нескольких асинхронных событий одного и того же типа, то система должна предусматривать некоторую диспетчеризацию AST-обработки. В частности, в МОС ВП организуется очередь AST-прерываний, т. е. все они обрабатываются последовательно в порядке возникновения, и их обработка приостанавливает выполнение основного процесса. Подчеркнем и тот факт, что существуют различные AST-очереди для каждого режима работы процессора. В системе возможно наличие четырех очередей для любого процесса. Они обрабатываются на

основе дисциплины циклического обслуживания с переходом от более привилегированного режима к менее привилегированному. Таким образом, AST-процедуры, выполняемые в режиме пользователя, могут быть прерваны для проведения AST-обработки режима ядра, в то время как последняя не может быть прервана.

Все системные обслуживающие программы, которые могут специфицировать AST-процедуры, должны иметь возможность оперировать 32-битовым AST-параметром. Такой параметр доступен AST-процедуре через стек. Если программист хочет использовать одну AST-подпрограмму для обработки событий от нескольких источников, то AST-параметр используется для их идентификации. Так, программа, взаимодействующая с несколькими терминалами, может выдать на каждый из них команду чтения, установив при этом AST-подпрограмму по завершению ввода. В этом случае AST-параметр должен содержать информацию, указывающую, на каком терминале завершилось чтение.

Управление AST-подпрограмме передается по макрокоманде CALL с указателем аргументов AP, содержащим ссылку на следующий список:



Параметр AST является единственным, который задается некоторой программой при спецификации AST-процедуры. Все остальное содержимое приведенного списка заполняется системой и может модифицироваться AST-процедурой.

Процедуры ввода-вывода кроме уже достаточно подробно рассмотренной службы QIO включают большое число обслуживающих подпрограмм. Во-первых, сюда относятся процедуры, обеспечивающие работу с логическими именами устройств. Эти процедуры позволяют создать, использовать и уничтожить логические имена, которые предназначены для уменьшения зависимости программ от конкретных внешних устройств. Логическое имя может быть поставлено в соответствие как отдельному устройству, так и устройству плюс каталогу и, возможно, файлу.

К этой же группе процедур относятся средства назначения канала и отмены назначения канала (логического пути между устройствами и процессом). Кроме QIO, собственно ввод-вывод организуют и другие процедуры. Они обеспечивают работу только на вывод или только на ввод, форматирование входных или выходных данных. Наконец, сюда же относятся процедуры занятия и освобождения устройств, получения информации об устройствах и каналах, а также работы с почтовыми ящиками.

Подпрограммы обработки исключительных ситуаций и средства изменения режима представляют собой набор процедур ядра, предназначенных для пользовательских подпрограмм обработки исключительных ситуаций, возникших при выполнении процесса, и для перехода пользовательского процесса в исполнитель-

ный режим либо режим ядра для выполнения специфицированных подпрограмм.

Интерфейс процедур ядра. В основе макровызовов системного обслуживания лежит использование макрокоманды CALL. Однако поскольку в большинстве случаев для выполнения требуемых функций должны быть заданы значения соответствующих параметров, то для передачи управления от пользовательского процесса в процедуру ядра применяются две формы макрокоманды: с использованием стека для хранения параметров CALLS (стековая форма) либо с построением специального списка CALLG для хранения параметров (списковая форма).

Для читателя-программиста очевидно, что раз речь идет об использовании микрокоманды CALL, то после того как программа, содержащая обращения к системным обслуживающим процедурам, оттранслирована и скомпонована, она уже содержит виртуальные адреса соответствующих точек входа. Здесь у читателя должен возникнуть естественный вопрос: что делать в том случае, когда перегенерируется сама МОС ВП? Ведь при этом возможно изменение виртуальных адресов некоторых процедур и, следовательно, возникает необходимость в перекомпоновке пользовательских программ. Читатель, знакомый с операционной системой ОС РВ, знает, что в ней именно так все и происходило. Однако в МОС ВП механизм общения с обслуживающими процедурами построен на совершенно другой основе. Суть его состоит в том, что первые страницы системного адресного пространства МОС ВП зарезервированы под векторы системного обслуживания, являющиеся специальными указателями на текущие виртуальные адреса обслуживающих процедур. По своему назначению эти векторы в общем-то аналогичны обычным векторам прерывания.

Для каждой системной обслуживающей процедуры в соответствующем векторе хранится некоторое число команд, обеспечивающих передачу управления на требуемый адрес. Иными словами, изменение расположения той или иной системной подпрограммы приведет лишь к изменению содержимого соответствующего вектора и никак не скажется на пользовательском процессе.

В частности, имя SYS$QIO представляет собой символический адрес вектора процедуры QIO.

Содержимое такого вектора в первую очередь определяется режимами, в которых выполняются вызывающая программа и обслуживающая процедура. Если режимы совпадают, то вектор содержит только 16-разрядную маску сохранения регистров и команду перехода. Так, если процедура имеет имя NAME, то ее вектор содержит следующую последовательность:

```
SYS$NAME:
        .WORD   ^M<R2,R3,R4,R5> ;Маска сохранения регистров
        JMP     EXE$NAME+2      ;Переход на обслуживание
```

В рассмотренном примере управление попадает на команду, расположенную в системном адресном пространстве по символическому адресу EXE$NAME+2. Такой метод вызова обеспечивает передачу управления на команду, отстоящую на два байта от входной точки обслуживающей процедуры, поскольку в ее начале в 2-байтовом поле запоминается маска сохранения регистров. После завершения обслуживания по команде возврата управление передается непосредственно пользовательской программе.

Если обслуживающая процедура и программа пользователя выполняются в различных режимах, то вектор содержит еще и команду изменения режима:

```
SYS$NAME:
        .WORD   ^M<R2,R3,R4>    ;Маска сохранения регистров
        CHMK    #CODE           ;Диспетчеризация и обслуживание
        RET                     ;Возврат в вызывающую программу
```

В данном случае по команде CHMK («Перейти в режим ядра») меняется режим процесса, а управление попадает в специальную программу-диспетчер. Последняя по значению кода CODE определяет требуемую системную обслуживающую процедуру и передает ей управление, предварительно скопировав содержимое стека вызывающей программы в стек вызываемой. Обслуживающая процедура завершает свое выполнение командой возврата. Управление при этом вновь попадает в программу-диспетчер, которая с помощью команды REI («Возврат из прерывания») изменяет режим процессора и передает управление на команду, следующую за CHMK, т. е. на команду возврата в вызывающую программу.

Остановимся теперь на реализации интерфейса программы, написанной на языке макроассемблера с процедурами ядра МОС ВП. При этом будем ориентироваться на стековую форму соответствующих макрокоманд, а также исходить из того, что все передаваемые в подпрограммы обслуживания параметры представлены в словах длинного формата и являются либо значениями параметров, либо указателями ячеек, в которых содержатся такие значения, либо, наконец, ссылками на адреса, куда процедура обслуживания должна записать результат обработки запроса. При необходимости обменяться с такой процедурой строкой данных пользователь в качестве входного (выходного) параметра задает адрес счетверенного слова — строкового описателя. Первое слово двойной длины при этом содержит длину строки, а второе — ее виртуальный адрес. Иными словами, для передачи строки достаточно указать только адрес строкового описателя. В том случае, если после обработки запроса процедура обслуживания будет возвращать в пользовательский процесс некоторую строку, то последний при формировании запроса должен предусмотреть передачу адреса такого строкового описателя. В первом его слове должна содержаться длина выделенного для формирования выходной строки буфера. Если его размер меньше реальной длины строки, то пользователю будет доступно лишь начало строки длиной, равной значению, указанному в описателе.

Макрокоманды системного обслуживания представляют собой собственно интерфейс для связи пользовательских процессов с процедурами обслуживания. В системной макробиблиотеке МОС ВП имеются соответствующие макроопределения в обеих формах: стековой и списковой. Как мы уже говорили, основное внимание будет уделено стековой форме, т. е. таким макроопределениям, обработка которых макроассемблером приводит к генерации макрокоманды CALLS.

Все макровызовы этих макрокоманд имеют следующую общую форму:

```
$NAME_S         KEYWORD1=Аргумент1
                KEYWORD2=Аргумент2


                KEYWORDN=АргументN
```

где NAME — имя макрокоманды; KEYWORD — ключевое слово; аргумент i (i∈[1, n]) — значение, передаваемое в процедуру обслуживания. В качестве примера приведем макровызов GETTIM («Получить время»):

```
$GETTIM_S       TIMEADR=BUFTIM
```

Его макрорасширение включает обращение к обслуживающей программе «Выдать время». Она возвращает в пользовательский буфер 64-разрядное число — внутреннее представление текущего времени суток. Это значение записывается в память по адресу BUFTIM, указанному пользователем. Приведенный макровызов послужит причиной генерации в пользовательской программе следующей последовательности команд:

```
PUSHAQ          BUFTIM
CALLS           #1, SYS$GETTIM
```

В макровызовах допустимо использование и других режимов адресации:

```
$GETTIM         TIMEADR=BUFTIM
$GETTIM         TIMEADR=(R9)
$GETTIM         TIMEADR=ARRAY[R5]
```

В последнем примере R5 должен содержать индекс текущего элемента массива времен ARRAY.

Коды возврата. Для оценки правильности обработки запросов в обслуживающих подпрограммах пользователю передаются специальные числовые значения, называемые кодами возврата. По соглашению все программы МОС ВП передают коды возврата в регистре R0. Возвращаемое значение не только идентифицирует правильность или неправильность выполнения пользовательского запроса, но и позволяет определить конкретную причину ошибки. Бит 0 кода определяет общий характер завершения операции: если он установлен, пользовательский запрос обработан правильно, в противном случае обнаружена ошибка. Следовательно, проверка успешности завершения работы процедуры может быть проведена с использованием лишь одной команды:

```
BLBC            R0,ERROR_POINT
```

или одного предложения на языке высокого уровня:

```
IF NOT SERVICE(PARAM)  THEN ERROR_POINT
```

Степень серьезности ошибки в коде возврата фиксируется в трех младших разрядах <2; 0>, и их значения имеют следующий смысл:

0 Предупреждение
1 Успешное завершение
2 Ошибка
3 Не используется
4 Серьезная ошибка
5—7 Не используются

Заметим, что все коды, идентифицирующие ошибки, имеют четные значения (бит 0=0), а идентифицирующие успешное выполнение — нечетные (бит 0=1). Кроме того, значение кода возврата, идентифицирующего ошибку, специфицирует

также системное сообщение, которое может быть получено с помощью системной программы $GETMSG («Выдать сообщение об ошибке»).

Как и в других операционных системах СМ ЭВМ, в МОС ВП для удобства программирования введены символические значения кодов завершения. Для определения их значений необходимо обратиться к описательной макрокоманде $SSDEF. Все символические значения кодов завершения имеют вид SS$__имя, например SS$__NORMAL – код нормального завершения, SS$__INSFARG — код ошибки «слишком много аргументов». Использование символических значений кодов значительно облегчает программирование, что можно продемонстрировать следующим примером:

```
$SSDEF                          ;Определить символические
                                ;значения кодов


.
<Вызов системной обслуживающей процедуры>
CMPL    #SS$_INSFARG,R0         ;Слишком много аргументов?
BEQL    SPEC_ERR                ;Переход, если да
.
.
.
```

6.7. ЯЗЫК КОМАНД ОПЕРАТОРА

Мы уже говорили о том, что язык оператора МОС ВП DCL, как и в большинстве других операционных системах современных ЭВМ, позволяет представить компьютер как устройство, предоставляющее весьма удобный интерфейс для программных средств, а ОС, следовательно, можно рассматривать как более высокий уровень архитектуры вычислительной системы, реализующий связь аппаратуры с пользователями и их программами. Как правило, любая развитая операционная система имеет несколько интерфейсов. Основными среди них являются: язык команд оператора, макрокоманды и подпрограммы системного обслуживания, языки программ-утилит, системы программирования. Рассмотрим первый из перечисленных интерфейсов и его реализацию в МОС ВП.

В современных интерактивных вычислительных системах язык команд оператора заменил широко распространенные в системах пакетной обработки языки управления заданиями. Используя терминал, с которого вводятся необходимые команды, пользователь имеет возможность оперативно вмешиваться в работу вычислительного комплекса. В новых машинах командный язык получает дальнейшее развитие, становится все ближе к естественному. Более того, опытный пользователь сейчас получил возможность расширять его в соответствии с требованиями конкретного применения.

Команды оператора МОС ВП по характеру выполняемых функций можно разделить на несколько основных групп:

1. Команды манипулирования файлами. Они обеспечивают копирование файлов, их переименование, стирание и другие операции над ними. Продолжив ранее начатое сравнение с системой ОС РВ, отметим, что в ней аналогичные функции реализовывались за счет запуска соответствующих программ-утилит.

2. Информационные команды обеспечивают оператора запрошенной информацией. Команды этой группы во многом схожи с информационными командами ОС РВ.

3. Управляющие команды предназначены для установки режимов работы терминалов, изменения текущего каталога, запуска пользовательских процессов, прекращения их выполнения и т. д. В том или ином виде подобные команды присутствуют в командном языке любой интерактивной операционной системы.

Работа с МОС ВП начинается с процедуры регистрации пользователя. При этом задаются идентифицирующие его данные: имя и пароль. Последний необходим для определения прав пользователя на доступ к системе. Выполняемый при регистрации диалог с МОС ВП имеет следующий вид[1]:

```
USERNAME:       USERX
.........
PASSWORD:       PAYROLLX        (Значение пароля, как правило, не
.........                        отображается на терминале)

        WELCOME TO MOC ВП
        ....... .. ... ..
$
```

Символ $ выдается интерпретатором команд МОС ВП при готовности к вводу и обработке очередной команды (в данном случае после завершения процедуры регистрации).

Для команд оператора МОС ВП используется следующий общий формат:

имя_команды параметр_1 параметр_2 ... параметр_*n*, где «имя_команды» представляет собой ключевое слово, задающее требуемое действие, а параметры — это обычно объекты, над которыми оно должно быть произведено.

Например, строка

```
$PRINT    MYFILE.LIS
```

представляет собой указание системе распечатать на системном устройстве печати файл с именем MYFILE.LIS.

Кроме того, в команде может быть указан один или более квалификаторов (ключей). Они могут задаваться как при ключевом слове, так и при любом параметре. Квалификатору всегда предшествует символ «/». Основное назначение ключей — задание дополнительной информации по выполнению затребованного действия.

---

[1] Здесь и далее подчеркнут текст, выводимый системой автоматически.

Так, команда

```
$PRINT/DELETE   MYFILE.LIS
```

вызывает печать содержимого файла MYFILE.LIS и его последующее удаление с устройства, на котором этот файл хранится. Наконец, возможна еще одна форма задания команд — ввод только имени команды. В ответ система запросит ввода требуемых для выполнения нужного действия параметров. Например,

```
$PRI
FILE: MYFILE.LIS
....
```

Здесь PRI — сокращенное имя команды, в ответ на ввод которого система запросила имя файла. Как видно из этого примера, МОС ВП допускает использование сокращений, но они не должны совпадать, т. е. каждое из них должно содержать столько символов, сколько необходимо, чтобы быть уникальным.

Поскольку одной из основных функций любой операционной системы является работа с файлами (МОС ВП в этом отношении не представляет исключения), то целесообразно для более полного понимания дальнейшего изложения ввести соглашения об идентификации файлов, принятые в изучаемой операционной системе.

В МОС ВП, как и в ОС РВ, имя файла имеет следующий формат: Имя.Тип;Версия.

Здесь *Имя* — дискриптивный идентификатор, содержащий от одного до девяти символов; *Тип* (иногда еще называется расширением) — трехсимвольный элемент, с помощью которого описывается тип хранящейся в файле информации; *Версия* — целое число, идентифицирующее историю создания и коррекции файла.

Стоит остановиться на одной весьма важной особенности идентификации файлов в МОС ВП (это, впрочем, полностью относится и к ОС РВ). Речь идет о типах файлов. Многие пользователи считают, что существуют типы файлов, запрограммированные в компонентах системы, и полностью убеждены в том, что не может существовать файл, содержащий исходный текст программы на языке Фортран с расширением, отличным от FOR. Это принципиально неправильно. Действительно, по умолчанию компилятор Фортрана принимает расширение FOR, однако если вы указали как тип файла три любых других символа, то именно с указанным расширением и будет найден файл. Поэтому, когда мы говорим о наиболее часто используемых обозначениях типов файлов, имеются в виду те, которые принимаются различными программами по умолчанию, и это не значит, что для файлов того же типа нельзя использовать иные расширения.

По умолчанию в МОС ВП обычно принимаются следующие расширения:

BAS Исходный текст программы на языке Бейсик
COM Файл, содержащий командную процедуру

DAT Файл с двоичными данными
DIR Файл каталога
EXE Файл, содержащий выполняемый образ программы
FOR Исходный текст программы на языке Фортран
LIS Файл, подготовленный для печати на АЦПУ
MAR Исходный текст программы на языке макроассемблера МОС ВП
OBJ Объектный файл
PAS Исходный текст программы на языке Паскаль

Что касается версии, то она в явном виде задается достаточно редко. Дело в том, что система ввода-вывода МОС ВП при создании или коррекции файла автоматически присваивает ему номер версии 1 или увеличивает существующий на 1 соответственно. По умолчанию любая утилита МОС ВП всегда использует последнюю версию. Иными словами, явное задание версии необходимо только при обращении к непоследней копии файла.

Приведем несколько примеров:

```
PHASER.OBJ;1
MTREKD.EXE
MTEKINI.MAR
ROBOT.FOR;6
```

Необычайно важную роль в системе играют специальные файлы, называемые каталогами. Запуск любой программы, обращение к любому файлу в МОС ВП осуществляется с указанием необходимого каталога, каждый из которых содержит набор некоторых элементов, соответствующих всем файлам пользователя или группы пользователей. Такой элемент описывает имя файла, его длину, адрес на диске, дату создания и другую информацию, требуемую файловой системе. Кроме того, любой из зарегистрованных в системе пользователей в каждый момент находится в некотором каталоге, т. е. как бы «прописан» в нем. Такой каталог называется «каталогом по умолчанию». В зависимости от привилегий пользователя он может «менять прописку» — переходить из одного каталога в другой, обращаться из текущего каталога к другим и т. д.

В любой момент пользователь с помощью команды DIR («Печать каталога») может получить информацию о своих файлах. Так, вводя просто команду DIR, мы запрашиваем у системы распечатку содержимого каталога по умолчанию. Вот один пример реакции МОС ВП на такую команду:

```
$DIR

DIRECTORY [GAMES]

ADVENT.EXE;1      DUNGEON.EXE;1     FILE.DAT;3        PRIMES.EXE;4
PROGRAM.EXE;2     PROGRAM.FOR;8     PROGRAM.OBJ;2
TICTAC.FOR;5
```

Для указания того, что файл расположен в некотором, отличном от текущего, каталоге, перед именем файла в квадратных скобках задается имя этого каталога, например:

```
[SYSGAMES]TICTAC.FOR
```

Полная спецификация файла МОС ВП, кроме того, включает описание соответствующего устройства и имя узла. Описание устройства состоит из имени устройства, имени контроллера и номера устройства. Так, имя, идентифицирующее тип устройства, может иметь следующие значения: TT (для терминала), LP (для устройства печати) и т. д. Как и в большинстве других операционных систем для СМ ЭВМ, в МОС ВП для указания устройства задается двоеточие, например диск BA0:.

Важной особенностью системы ввода-вывода МОС ВП является то, что она в состоянии выполнять операции над удаленными файлами, т. е. над теми, которые расположены на некотором, удаленном от рассматриваемого, комплексе СМ1700. В случае, если на обе ЭВМ загружено сетевое программное обеспечение, то стандартно обеспечивается доступ к удаленным файлам. Так, по команде

```
$TYPE   NODE2::DJA1:[MANAGER]NEW.COM
```

на терминал, с которого задана команда, будет выведено содержимое файла NEW.COM, расположенного на диске DJA1 в каталоге MANAGER удаленной СМ1700, имеющей имя узла NODE2. Как следует из приведенного примера, имя узла выделяется двойным двоеточием.

Таким образом, полная спецификация файла МОС ВП имеет следующий формат:

[Узел::] [Устройство:] [Каталог] Имя [.Тип [;Версия]]

В спецификации обязательно только собственно имя файла, для остальных ее параметров приняты соответствующие умолчания: узел — тот, на котором выполняется команда, содержащая спецификацию файла; устройство — системное устройство SYS$DISK; каталог — текущий; тип файла — принятый по умолчанию в обслуживающей программе; версия — последняя. Разумеется, в зависимости от конкретных условий часто приходится в явном виде задавать не только имя файла, но и другие составляющие его спецификации.

Для более эффективной организации доступа к своим файлам пользователь МОС ВП может создавать подкаталоги, образующие древовидную иерархическую структуру с максимальным числом уровней иерархии 9. Пример такой файловой структуры приведен на рис. 6.15. Как и каталоги пользователей, подкаталоги представляют собой файлы специального типа (с расширением DIR). Они являются элементами каталогов, а последние, в свою очередь, составляют главный системный каталог. Приведем несколько примеров:

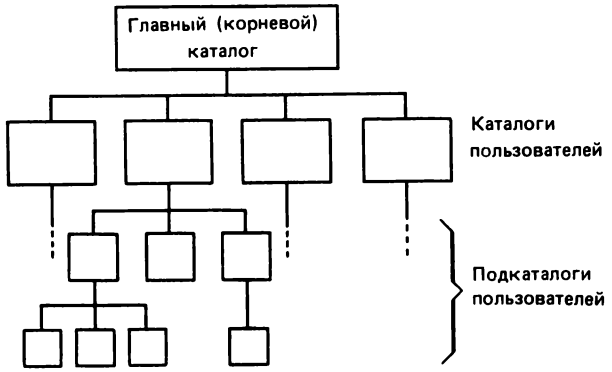


Рис. 6.15. Каталоги в системе МОС ВП

```
[NETWORK]
[NETWORK.PROCESS]
[NETWORK.PROCESS.SOURCE]
[NETWORK.PROCESS.COMMAND]
```

Следовательно, в приведенной выше полной спецификации файла МОС ВП вместо каталога может быть указана часть иерархического дерева.

Для упрощения работы с данными в системе используется большое число сокращений. Так, файл можно искать не только по явно указанному имени, но и по имени, заданному, например, в следующем виде: AB*CE.*;*. Это означает, что имя файла начинается с букв AB, а заканчивается буквами CE, а между ними допускается любая комбинация символов. Более того, в данном примере адресуются все такие файлы независимо от расширения и номера версии (этот факт отражается в записи как «*;*»). Подобным образом можно указывать и каталоги. Так, для задания подкаталогов текущего каталога достаточно вставить в запись символ «.». Например:

```
$SET DEF [NETWORK.PROCESS]
$DIR [.COMMAND]
DIRECTORY DBA2:[NETWORK.PROCESS.COMMAND]
NETCONFIG.COM;1    DISDTR.COM;3    NETTEST.COM;1
```

В данном примере по команде SET DEF (сокращение от DEFAULT) устанавливается значение «каталог по умолчанию». Далее, по команде DIR (DIRECTORY) затребована распечатка содержимого каталога [NETWORK.PROCESS.COMMAND].

Таблица 6.6. **Команды манипулирования файлами**

Имя команды	Аргумент	Функция
APPEND	Входной файл 1, входной файл 2, ...	Добавление некоторого числа файлов в конец первого файла
COPY	Входной файл␣выходной файл	Копирование содержимого входного файла в выходной
CREATE/DIR	Имя каталога	Создание каталога или подкаталога

Окончание табл. 6.6

Имя команды	Аргумент	Функция
DELETE	Файл 1, файл 2, ...	Удаление указанных файлов
DIRECTORY	Имя каталога	Печать содержимого указанного каталога или каталога, используемого по умолчанию (если имя не задано)
PRINT	Файл 1, файл 2, ...	Печать содержимого заданных файлов на системном устройстве печати
PURGE	Файл	Стирание всех, кроме последней, версий файла
RENAME	Старое имя ␣ новое имя	Переименование файла
TYPE	Файл	Печать содержимого файла на терминале, с которого введена команда

Подобные упрощения и ряд других МОС ВП допускает при работе с файлами и каталогами.

Приведенный пример демонстрирует возможность использования не полных названий команд и их операндов, а некоторых сокращений. Рассматриваемая операционная система реализована таким образом, что в ней практически во всех случаях возможно применение сокращений вместо полных слов. При этом существуют два ограничения: сокращение может включать набор произвольной длины следующих друг за другом, начиная с первого, символов исходного слова. Любое сокращение должно быть уникальным.

Поскольку значительная часть выполняемой операционной системой работы заключается в создании файлов и оперировании ими, то, пожалуй, едва ли не чаще других в МОС ВП используются команды манипулирования файлами. Состав и функции основных из них приведены в табл. 6.6, а на рис. 6.16 показано типовое применение.

Как уже отмечалось, в МОС ВП имеется набор информационных команд, необходимых для получения информации о состоянии процесса, системы в целом или некоторого ее компонента. Особо здесь следует выделить команду HELP, с помощью которой пользователь может получить информацию об использовании любых команд или программ МОС ВП. Доступ к HELP-информации организован на иерархической основе: на первом

```
$SHO DEF                    !Запросить каталог, используемый по
                            !умолчанию
$DIR                        !Просмотреть список файлов этого каталога
DIRECTORY DJA1:[OLD]
GRAPH.EXE;3 GRAPH.FOR;21 GRAPH.LIS;1 GRAPH.OBJ;1
$CRE/DIR [OLD.SOURCES]      !Создать подкаталог для исходных текстов
$COP GRAPH.FOR [.SOURCES]   !Копировать исходный файл в подкаталог
$DEL GRAPH.FOR;21           !Стереть файл из основного каталога
$SET DEF [.SOURCES]         !Установить новый подкаталог как
                            !используемый по умолчанию
$DIR                        !Просмотреть список файлов этого каталога
DIRECTORY DJA1:[OLD.SOURCES]
GRAPH.FOR;1
```

Рис. 6.16. Типовое применение команд манипулирования файлами

уровне доступна общая информация об указанной команде, после этого выводится список аргументов выбранной команды, далее описание заданного аргумента и т. д. На следующем примере дан типовой формат одноуровневой HELP-информации:

```
$HELP TYPE
      TYPE

            Отображает содержимое одного или группы
            файлов на текущее выводное устройство.
            Формат:

            TYPE спецификация-файла

            Доступная дополнительная информация:
            Квалификаторы параметров
            /OUTPUT — спецификация-файла
```

Второй наиболее общей командой этой группы является команда SHOW. Она обеспечивает вывод на пользовательский терминал или на системную консоль информации о процессах, их статусах, об используемых ими устройствах и т. д. В табл. 6.7 приведены некоторые часто используемые аргументы команды SHOW.

Таблица 6.7. **Основные аргументы команды SHOW**

Аргумент	Функциональное назначение
DAYTIME	Выдать текущую дату и время
DEFAULT	Выдать используемое по умолчанию устройство и цепочку каталогов
DEVICES	Выдать сообщение о состоянии устройств, используемых системой
PROCESS	Выдать информацию о состоянии пользовательского процесса
STATUS	Выдать состояние пользовательского процесса, когда в нем выполняется образ
SYSTEM	Выдать состояния всех процессов, существующих в системе
TERMINAL	Выдать характеристики терминала

После освоения программистом командного языка, когда ему становятся очевидными все его преимущества, как правило, возникает желание завести собственные умолчания, создать свой набор команд, отличный от системного. Все это можно сделать следующими способами:

1. Выбором синонимов, применяемых вместо имен команд или командных строк.
2. Заданием значений квалификаторов параметров (квалификаторы в МОС ВП соответствуют ключам в ОС РВ) по умолчанию.
3. Созданием командных процедур, обеспечивающих реализацию заданного набора команд.

Рассмотрим, как с помощью перечисленных способов можно создать свой микроязык. Для этого обратимся к возможностям языка интерпретатора системы.

Так, оператор назначения дает возможность присваивать некоторым символам числовые или текстовые значения. Например, вместо того, чтобы набирать на клавиатуре строку

```
$SHOW TIME
```

воспользуемся оператором назначения вида

```
$TIM:==SHOW TIME
```

После такого определения по команде

```
$TIM
```

будет получен тот же результат, который даст исходная команда.

Аналогичным образом учитываются и квалификаторы:

```
$PD:==PRINT/DEL
$PD FLE.TXT,FLE2.DOC
```

В том случае, когда пользователю часто требуется выполнение некоторых модификаций одних и тех же последовательностей команд, целесообразно использование командных процедур. Такая процедура является файлом, в который занесена требуемая последовательность команд и при необходимости данных. Отметим, что реализуемые в ОС РВ косвенные командные файлы призваны выполнять те же функции, что и командные процедуры, но последние имеют несравненно более широкие возможности.

Пусть требуется часто транслировать с макроассемблера, компоновать и запускать на выполнение программу SQUAROOT. Для этого можно создать следующую командную процедуру:

```
MACRO/LIST  SQUAROOT
LINK SQUAROOT
RUN  SQUAROOT
```

Если такой файл назвать EXE.COM, то выполнение командной процедуры может быть достигнуто вводом набора символов

```
$@EXE
```

где знак @ оповещает интерпретатор команд, что требуется запустить командную процедуру (как уже отмечалось, по умолчанию принимается, что расширение файлов, содержащих такие процедуры, имеет вид «COM»). Каждую команду, содержащуюся в файле EXE.COM, интерпретатор обрабатывает так, как если бы она была введена с терминала.

Хотя приведенная командная процедура и упрощает взаимодействие пользователя с системой, тем не менее у нее есть существенный недостаток: командная процедура оперирует только программой на макроассемблере SQUAROOT. Подобный недостаток легко устранить, если организовать передачу параметров

командной процедуре при ее запуске. В МОС ВП это реализовано с помощью параметрической структуры командных процедур: в них допускается использование формальных аргументов с именами P1 — P8. Замена формальных аргументов на значения фактических производится в момент запуска процедуры. С учетом сказанного процедура EXE может быть записана так:

```
MACRO/LIST 'P1'
LINK 'P1'
RUN 'P1'
```

Теперь для трансляции, компоновки и запуска программы SQUAROOT мы должны ввести строку

```
$@EXE SQUAROOT
```

При обработке процедуры интерпретатор команд осуществляет замену формального параметра, записанного в апострофах, на значение фактического, в данном случае на SQUAROOT. Как легко может убедиться сам читатель, подобная форма записи командной процедуры обеспечивает выполнение рассмотренных действий над любой программой.

Теперь воспользуемся уже известным нам механизмом назначения:

```
$EXE:==@EXE
```

Тогда для трансляции, компоновки и запуска на выполнение программы STARTREK необходимо лишь ввести строку вида

```
$EXE STARTREK
```

Иными словами, используя рассмотренные средства, мы создали свою собственную команду.

Можно с уверенностью предположить, что одним из первых мероприятий, выполняемых постоянным пользователем МОС ВП, будет создание командной процедуры LOGIN. Дело в том, что после выполнения процедуры регистрации система ищет в каталоге, определенном при регистрации для данного пользователя, файл LOGIN.COM и при успешном поиске автоматически запускает содержащуюся в нем процедуру. Расширение такого файла в целях занесения в него пользовательских символов и значений, принимаемых по умолчанию, представляет собой весьма эффективное средство адаптации интерфейса МОС ВП к требованиям конкретного применения. В частности, файл LOGIN.COM может содержать следующие команды:

```
TIME:==SHOW TIME
EXE:==@[NEW.PASCAL]EXE
MAIL:==RUN [SYSTEM]MAIL
TYPE [SYSTEM]DAILY.DAT
SET DEF [NEW.PASCAL.SOURCE]
```

В данном случае после регистрации пользователя для него в качестве каталога по умолчанию будет принят [NEW.PASCAL.SOURCE], на терминал будет выведено содержимое файла DAILY.DAT и определены три новые команды: TIME — печать времени, EXE — запуск рассмотренной выше процедуры EXE.COM из каталога [NEW.PASCAL] и MAIL — запуск программы «почта».

6.8. УТИЛИТЫ ДЛЯ РАЗРАБОТКИ ПРОГРАММ

Рассмотрим теперь другой тип интерфейса, предоставляемого пользователям МОС ВП. Речь пойдет о программном обеспечении для разработки программ, включающем такие утилиты, как редакторы, компоновщики, отладчики, трансляторы и другие средства, которые иногда объединяются под названием систем программирования.

Процесс подготовки программы к эксплуатации включает несколько этапов и может быть представлен графически так, как это сделано на рис. 6.17. Здесь отражены именно те средства программного обеспечения МОС ВП, которые рассматриваются в данной книге.

Редактор. Это компонент МОС ВП, используется для создания и коррекции исходных текстов программ и для работы с текстовыми файлами. Поскольку редактор представляет собой одну из наиболее часто применяемых программ, его существенной характеристикой является простота. Нередко разные пользователи предпочитают различные редакторы и каждый из них считает, что именно тот, который применяет он, наиболее удобен и эффективен.

Хотя редакторы весьма сильно отличаются друг от друга, они, тем не менее, имеют следующие общие особенности:

печать или отображение на экране терминала нескольких следующих друг за другом строк файла;

поиск заданного набора символов по всему файлу;

вставка строки символов в любое место внутри файла;

стирание строк;

замена одних строк символов на другие;

перемещение указателя текущей позиции в пределах файла;

завершение сеанса редактирования с созданием нового файла, содержащего все изменения и добавления.

Большинство наиболее популярных редакторов допускают протоколирование своей работы как на терминалах типа телетайп, так и на видеотерминалах. Именно различия в этих устройствах определили разделение редакторов на строчные и экранные.

При работе со строчным редактором пользователь оперирует последовательностью смежных строк файла, подлежащих изменениям. Такой режим является естественным и единственно возмож-

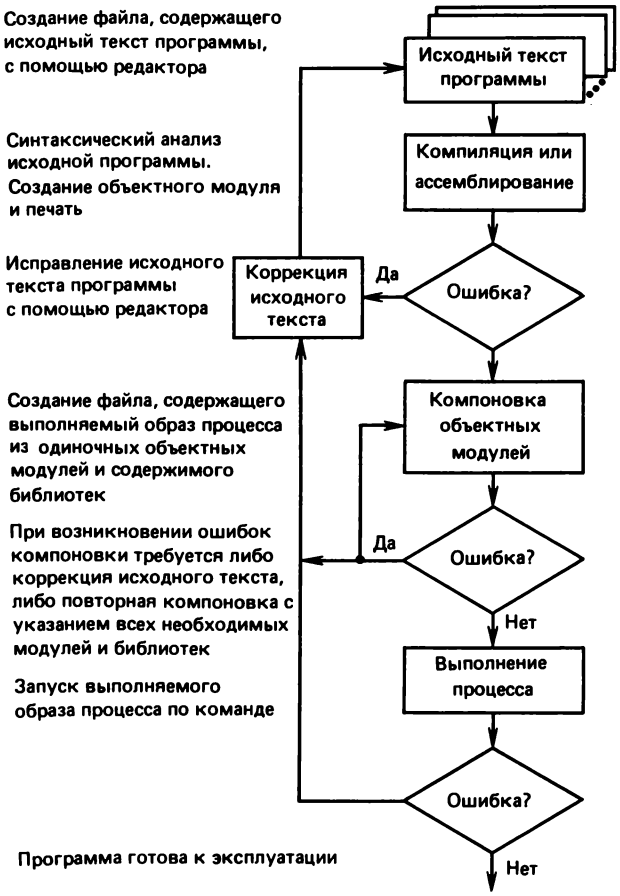


Рис. 6.17. Этапы разработки программы

ным при работе на телетайпах и других печатающих терминалах. В экранных или оконных редакторах, функционирующих лишь на видеотерминалах, в любой момент на экране отображается некоторый участок — «окно» файла. С помощью курсора (маркера) пользователь может задавать место в окне, куда должно быть внесено изменение. Специальные команды редактора позволяют перемещать маркер в любом направлении, сдвигать его на требуемое число символов, слов, строк, параграфов и т. д. Если необходимо вставить в файл некоторые текстовые элементы, то по мере их набора на клавиатуре они заносятся в окно, перемещая в нем или даже удаляя из него последующий текст. Обычно последовательность отображаемых и, следовательно, доступных для обработки строк файла (окно) имеет размер экрана используе-

мого дисплея. Как правило, современные редакторы ориентированы на аппаратные возможности интеллектуальных терминалов, самостоятельно выполняющих перемещения курсора и даже ряд простых команд редактирования. Все это существенно снижает нагрузку на центральный процессор.

Транслятор и компилятор. После создания файла, содержащего текст исходной программы на некотором языке программирования, пользователь должен выполнить ее трансляцию или компиляцию и создать так называемый объектный файл, т. е. файл, содержащий объектный модуль. Процесс создания объектных кодов с помощью двухпроходного ассемблера мы уже рассмотрели в гл. 2. Теперь подойдем к обсуждению других, связанных с этим, вопросов с учетом того, что содержимое объектного файла является исходной информацией для компоновщика.

Во-первых, поскольку в любой развитой операционной системе поддерживаются различные языки программирования, объектный файл представляет собой набор записей, выполненных в некоторых форматах, единых для любых языков. Иными словами, все трансляторы и компиляторы, воспринимая исходные тексты программ, составленных в соответствии с правилами тех или иных языков, переводят их в некоторый единый вид, который понимает системная программа — компоновщик. Во-вторых, объектный модуль не может состоять лишь из двоичных кодов, поскольку компоновщику необходима информация для объединения различных модулей и коррекции адресов. Рассмотрим это несколько подробней.

Одна из целей модульного программирования заключается в предоставлении возможности раздельной подготовки и независимой автономной отладки различных частей одной большой и сложной программы. Соответственно вместо одного большого исходного модуля создается набор небольших блоков исходного текста (возможно, даже на различных языках программирования). Каждый из них включает одну или более подпрограмм, которые могут обращаться к другим, расположенным вне пределов данного модуля. Естественно, что в этом случае транслятор или компилятор не может сгенерировать полную машинную программу, поскольку им неизвестны адреса и символьные значения, определенные не в данном блоке. Они не знают также, где именно в виртуальной памяти будет размещен создаваемый объектный код. Таким образом, объектный модуль представляет собой программную структуру, которая в большинстве операционных систем не допускает выполнения, и некоторые ее элементы подлежат изменениям в процессе обработки компоновщиком.

Компоновщик. Этот компонент операционной системы, как правило, является самой сложной из программ, служащих для создания и отладки программного обеспечения. И компоновщик МОС ВП не является исключением.

Компоновщик предназначен для объединения раздельно оттран-

слированных и скомпилированных модулей в образ исполняемого процесса. При этом им выполняются все необходимые действия по разрешению символических ссылок между модулями и по распределению виртуальной памяти программы. При создании объектного модуля транслятору или компилятору неизвестен адрес памяти, в которой будет выполняться программа, т. е. такие модули, как правило, перемещаемы. Поскольку в СМ1700 допустимы позиционно-независимое кодирование и относительная адресация, следовательно, решение задачи перемещаемости обеспечивается. Компоновщик МОС ВП анализирует требования каждого модуля и с их учетом формирует виртуальный образ процесса. Каждый из объединяемых модулей при этом может включать некоторое число программных секций с самостоятельными значениями атрибутов. В процессе компоновки осуществляется группировка секций со сходными атрибутами, в частности секций, для которых разрешено только чтение, допустимы и чтение, и запись, и все они помещаются в общее адресное пространство. После этого компоновщик осуществляет разрешение адресных ссылок между модулями.

Помимо разрешения ссылок между модулями, сформированными пользователем в явном виде при написании исходной программы (например, в виде оператора CALL ABSOL(A) на языке Фортран), компоновщик удовлетворяет также ссылки, порожденные транслятором (например, при трансляции Фортран-оператора WRITE (...)). В последнем случае ссылки, как правило, удовлетворяются из библиотек, представляющих собой в МОС ВП наборы объектных файлов макроопределений. В начале каждой библиотеки размещается словарь элементов, упрощающий и ускоряющий поиск в ней. В том случае, когда при создании образа процесса некоторые ссылки не могут быть удовлетворены из всех указанных пользователем библиотек, компоновщик автоматически переходит к просмотру системной. В ней имеется целый ряд необходимых подпрограмм обслуживания. Сюда же могут быть включены модули исполнительных систем таких языков, как Фортран или Паскаль.

На заключительном этапе построения процесса компоновщик формирует специальный файл, в который и записывается результат компоновки. Из файла образа впоследствии программа загружается в операционную память для исполнения. Содержимое такого файла включает исполняемую программу, а также информацию, необходимую системной службе управления памятью: описание секций программы, их атрибуты, размеры, адреса в виртуальном адресном пространстве, месторасположение в самом файле. Все эти данные используются при построении таблицы страниц процесса.

Отладчик. Каждому программисту хорошо известна справедливость афоризма из программистского эпоса: «Любая последняя найденная в программе ошибка является ошибкой предпоследней».

Действительно, практически ни одна реальная программа не начинает работать сразу после ее написания независимо от тщательности проработки. Отладка — это один из наиболее важных, сложных и трудоемких этапов разработки программного обеспечения, и поэтому хороший отладчик всегда является весьма ценной программой. На сегодня можно выделить несколько типов отладчиков. Некоторые из них компонуются вместе с отлаживаемой программой (как, например, в ОС РВ), другие функционируют как отдельные независимые процессы. Однако все отладчики в том или ином объеме реализуют следующие основные функции:

1. Проверку и(или) установку значения содержимого указанных пользователем ячеек памяти. Это касается как ячеек с кодами команд, так и тех, которые содержат данные.
2. Задание адресов точек останова, т. е. адресов команд, при достижении которых отлаживаемая программа должна остановиться.
3. Организацию шагового режима работы программы, т. е. ее останова после выполнения каждой инструкции.
4. Осуществление арифметических вычислений, нахождение значений выражений.
5. Задание контрольных точек.

Сказанное в полной мере относится и к отладчику МОС ВП.

Глава 7. РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ

7.1. ВЫБОР ЭЛЕМЕНТНОЙ И КОНСТРУКТИВНОЙ БАЗЫ

Основу элементной базы вычислительного комплекса СМ1700 составляют микросхемы программируемой матричной логики (ПМЛ) и микропроцессорные секции серии К1804. Широко используются для реализации памяти микропрограмм различных модулей комплекса программируемые постоянные запоминающие устройства (ППЗУ) и запоминающие устройства с произвольной

выборкой (ЗУПВ). Емкость используемых микросхем ППЗУ и ЗУПВ колеблется от 2 до 16 кбит в корпусе одной микросхемы. Блок оперативной памяти вычислительного комплекса СМ1700 построен на микросхемах динамических ЗУПВ емкостью 64 кбит в корпусе. Кроме того, в комплексе используются микросхемы различной степени интеграции быстродействующих серий К531 и К555.

Однако основными элементами СМ1700 следует все-таки считать микросхемы ПМЛ. Микросхема ПМЛ — это программируемая пользователем (т. е. изготовителем технических средств СМ1700) логическая матрица, представляющая совокупность матриц И и ИЛИ и эквивалентная нескольким малым и средним микросхемам, размещенным на одном кристалле. Обычно одна микросхема ПМЛ замещает 4—5 корпусов обычных микросхем и тем самым обеспечивает общее сокращение числа интегральных схем в 4—5 раз.

Исходная микросхема ПМЛ содержит плавкие перемычки, и перед установкой на плату или блок элементов она программируется на специальной установке, называемой программатором, по конкретной таблице. При программировании пережигаются определенные плавкие перемычки. В качестве программатора используются такого же типа установки, как и для программирования ППЗУ.

Достоинства ПМЛ заключаются не только в общем сокращении числа микросхем, но и в ряде других факторов. Среди них — гибкость и возможность ускорения разработки схем, возможность легкого внесения изменений в схему без проведения изменений в ее других частях и перетрассировки платы, высокое быстродействие вследствие использования при их изготовлении технологии диодов Шотки.

При построении схем, особенно с нерегулярной логикой, микросхемы ПМЛ имеют определенные преимущества перед полузаказными вентильными матрицами, ППЗУ и программируемыми логическими матрицами (ПЛМ).

Полузаказные вентильные матрицы представляют собой большие интегральные схемы (БИС), и их использование позволяет существенно сократить объемы аппаратных средств. Однако их разработка и производство требуют высокоразвитых систем автоматизации проектирования, и внесение любого, даже незначительного изменения потребует нового изготовления данной вентильной матрицы у производителя микросхем.

В случае ПМЛ любое изменение в таблицу их программирования может оперативно вносить сам пользователь. Эта оперативность облегчает также ремонт и быструю замену вышедших из строя микросхем ПМЛ.

Проведем теперь небольшое сравнение микросхем ПМЛ с микросхемами ППЗУ и ПЛМ. Все они отвечают концепции программирования и могут быть представлены как совокупность матриц И или ИЛИ.

Микросхемы ППЗУ — это логические схемы из элементов И — ИЛИ с фиксированной матрицей И и программируемой ИЛИ, т. е. в ППЗУ на матрице И задействованы все рабочие термы (известные больше как адреса программируемой памяти). По каждому из этих термов-адресов можно программировать любую информацию. Совокупность содержимого, запрограммированного по каждому адресу, и составит матрицу ИЛИ. Недостатком микросхем ППЗУ является их ограниченная гибкость при построении схем с нерегулярной логикой.

В ПЛМ обе матрицы И — ИЛИ имеют плавкие перемычки, и, следовательно, они обе программируются в соответствии с требуемыми логическими уравнениями. Недостатком этих микросхем является сложность программирования. Большого распространения в технических средствах СМ ЭВМ они не получили.

Концепция построения микросхем ПМЛ заключается в наличии в них программируемой матрицы И и фиксированной матрицы ИЛИ. Поэтому, чтобы получать различные комбинации логических функций И — ИЛИ, используются схемы ПМЛ различных типов, отличающихся конфигурацией схем ИЛИ. Таким образом, определение соединений матрицы ИЛИ является задачей выбора соответствующего типа микросхемы, а не задачей программирования. При таком подходе исключается необходимость во второй матрице с плавкими перемычками при малых потерях общей гибкости.

Схемы ПМЛ в отличие от ПЛМ могут иметь на выходе регистры на различное число разрядов. Выходы регистров могут в качестве обратной связи соединяться с входами ПЛМ. Обратные связи позволяют строить на ПМЛ автоматы (последовательностные схемы), которые способны запоминать свое предшествующее состояние и изменять свою функцию в зависимости от этого состояния.

Другая возможность, имеющаяся у микросхем ПМЛ,— это совмещение входных и выходных сигналов, т. е. один и тот же сигнал может быть как входным, так и выходным в зависимости от программирования матрицы И и значения соответствующих сигналов управления.

В вычислительном комплексе СМ1700 используются 4 различных типа микросхем ПМЛ: один из них чисто комбинационный, а три — регистровых. Комбинационный тип имеет 10 входов и 8 выходов, 6 из которых являются совмещенными. Регистровые типы имеют регистровые выходы на 4, 6 и 8 разрядов. Регистры построены на D-триггерах.

Общее количество микросхем ПМЛ, используемых в процессорном блоке СМ1700, достигает 105.

Другим важным компонентом элементной базы, как уже отмечалось, является серия К1804, основу которой составляет микропроцессорная секция К1804ВС1 (рис. 7.1). Элемент К1804ВС1 представляет собой 4-разрядный микропроцессор и состоит из двухпортового ЗУПВ, содержащего 16 рабочих регистров, а также из Q-регистра, арифметико-логического устройства (АЛУ), сдвигателей и мультиплексоров.

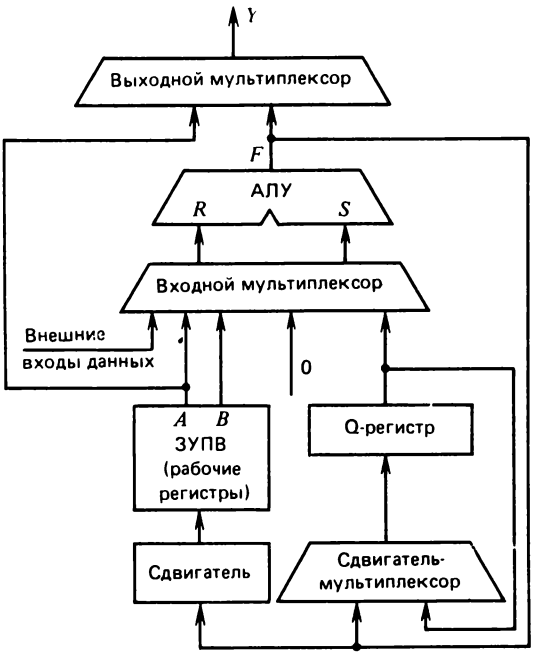


Рис. 7.1. Блок-схема микропроцессорной секции К1804ВС1

Рабочие регистры представляют собой местную память микропроцессора, где могут запоминаться как промежуточные, так и окончательные результаты арифметических и логических операций.

Устройство АЛУ является той частью микропроцессора, которая выполняет арифметические и логические операции в соответствии с управляющими сигналами, поступающими на входы микропроцессора (как правило, из управляющей памяти микрокоманд). В АЛУ на входы R подключаются выходы с мультиплексора, на входы которого подсоединяются внешние входы данных, выходы порта А ЗУПВ и входы установки нуля. В АЛУ на входы S через мультиплексор можно подать выходы портов А и В, Q-регистра и установку нуля.

Выходные данные (*F*) из АЛУ могут быть направлены в Q-регистр или ЗУПВ, а также через двухвходовой мультиплексор (на другой вход которого поступает выход порта *A* ЗУПВ) — на выходные передатчики (на рис. 7.1 не показаны) и на выход *Y*. Дешифратор функций АЛУ определяет, логическая или арифметическая функция должна быть выполнена, а дешифратор приемника результата — куда должен быть направлен результат: внутрь микропроцессора или за его пределы (на внешнюю шину).

Загружается Q-регистр с выходов АЛУ и используется как регистр для временного запоминания данных. Сигналы с выхода Q-регистра могут через АЛУ поступать на вход самого себя, а также сдвигаться вправо или влево.

Объединение восьми таких микропроцессорных секций образует 32-разрядные пути данных центрального процессора.

В процессоре операций с плавающей точкой устанавливается 20 таких секций, образуя 80-разрядную сетку обработки чисел с плавающей точкой различной точности.

Выше была рассмотрена элементная база, составляющая основу вычислительного комплекса СМ1700. Теперь кратко остановимся на его конструктивном исполнении.

Минимальная конфигурация СМ1700 включает процессорный блок и 1—2 дисковых накопителя (рис. 1.1). Процессорный блок представляет собой комплектный блок с источником питания. Он содержит несколько функциональных модулей:

центральный процессор (ЦП) с консольной подсистемой, состоящей из трех плат (блоков элементов);

процессор операций с плавающей точкой на одной плате;

интегрированный контроллер диска на одной плате;

оперативную память емкостью до 5 Мбайт (по 1 Мбайту на плате);

многофункциональный контроллер связи на одной плате для выхода на асинхронные и синхронные каналы. При расширении минимальной конфигурации комплекса используется блок расширения системы. В него вставляются контроллеры внешних устройств, а также дополнительные многофункциональные контроллеры связи. Кабелем интерфейса «Общая шина» (ОШ) этот блок подключается к процессорному блоку.

Размер печатных плат, используемых в вычислительном комплексе СМ1700, $411{,}2 \times 220$ мм.

7.2. РЕАЛИЗАЦИЯ ПРОЦЕССОРА

Центральный процессор вычислительного комплекса СМ1700 может выполнять полный набор инструкций, включая инструкции плавающей арифметики, а также работать в режиме совместимости с вычислительными комплексами семейства СМ4. Это позволяет пользовательским программам этого семейства работать без изменения. В режиме совместимости могут использоваться только непривилегированные инструкции СМ4, т. е. инструкции, которые разрешены для пользователя.

Вычислительная система СМ1700 имеет:

объем виртуальной памяти 4 Гбайт;

объем физической оперативной памяти от 1 до 5 Мбайт, причем максимальный объем определяется конструктивными ограничениями, а не архитектурными;

шестнадцать 32-разрядных общих регистров;

32 уровня приоритетных прерываний;

диспетчер памяти с буфером трансляции, позволяющим минимизировать время преобразования виртуальных адресов в физические адреса оперативной памяти;

буфер инструкций, позволяющий выбирать следующую инструкцию на фоне исполнения текущей (буфер имеет длину одного двойного слова и всегда поддерживается полным);

процессор операций арифметики с плавающей точкой (ППТ), обрабатывающий форматы данных с одинарной F, двойной D точностью, а также форматы с расширенным порядком G и учетверенной точности с расширенным порядком H;

таймер и «годовые часы»;

консольно-диагностический процессор (консольная подсистема), управляемый микропроцессором, с выходом на кассетный магнитофон (устройство начальной загрузки), консольный терминал и дистанционную диагностику;

системный интерфейс ОШ для подключения внешних устройств.

На рис. 1.1 приведена блок-схема СМ1700.

Центральный процессор (ЦП) состоит из трех блоков элементов (БЭ): арифметико-логического процессора (АЛП); консольно-диагностического процессора (КДП); контроллера памяти (КП).

Процессор КДП управляется 8-разрядным микропроцессором и имеет выход на три асинхронные дуплексные линии для связи с консольным терминалом (КТ), сдвоенным кассетным магнитофоном (устройство начальной загрузки) и со средствами дистанционной диагностики. Микропроцессором управляют программы, которые записаны в программируемой постоянной памяти ППЗУ емкостью от 4 до 6 Кбайт и в перезаписываемой памяти ЗУПВ емкостью 16 Кбайт. В ППЗУ хранятся программы начального запуска системы и программы диагностики, а в ЗУПВ — программы управления консольным терминалом, кассетным магнитофоном, программы связи с другими блоками процессора, а также диагностические программы. Все программы в ЗУПВ заносятся с кассетной ленты, устанавливаемой в магнитофон, основное назначение которого — запись в управляющую память центрального процессора микропрограмм системы команд или микропрограммной диагностики.

Блок элементов АЛП выполняет арифметические и логические операции, необходимые для реализации набора инструкций. Пути данных АЛП содержат память объемом 256 ячеек по 32 разряда, которая имеет общие, привилегированные и рабочие регистры центрального процессора. Управление путями данных осуществляется с помощью устройства микропрограммного управления, которое состоит из микросеквенсера (узла формирования адреса микрокоманды) и управляющей памяти емкостью 16К ячеек 24-разрядных микрокоманд, построенной на микросхемах ЗУПВ.

Связь с оперативной памятью (ОП) и ОШ осуществляется через БЭ КП. Обращение к ОП инициализируется или микропрограммой процессора, или запросами прямого доступа со стороны внешних устройств на ОШ. Блок элементов КП содержит схемы

формирования физического адреса памяти, арбитр ОШ, ротатор данных (для перепозиционирования данных), свои собственные микросеквенсер и управляющую память, которые предназначены для управления платой КП. Интерфейс ОШ подключается к плате КП, за исключением 16-разрядных линий данных, которые подсоединяются к плате КДП. Плата КП объединяется с двумя другими платами центрального процессора 32-разрядной шиной данных.

К центральному процессору подключаются также БЭ интегрированного контроллера диска (ИКД) и процессора операций арифметики с плавающей точкой (ППТ). Для управления ИКД и поддержки пользовательского микропрограммирования устанавливается дополнительная управляющая память объемом 4К микрокоманд. Микропрограммы загружаются в управляющую память во время процедуры первоначальной загрузки. К ИКД может подсоединяться до четырех дисковых накопителей. Передача данных между ИКД и центральным процессором производится по 32-разрядной шине и управляется микропрограммой центрального процессора. Для передачи 32-разрядного слова между процессором и ИКД со стороны последнего выставляется микропрерывание. Для вре́менного хранения данных ИКД имеет память типа FIFO («первым вошел — первым вышел») емкостью 1 Кбайт. Прерывания, не связанные с передачей данных, осуществляются через ОШ по уровню 3П5.

Процессор ППТ представляет собой процессор, работающий параллельно с центральным процессором и предназначенный для ускорения выполнения инструкций арифметики с плавающей точкой и некоторых инструкций целочисленной арифметики.

Здесь необходимо отметить, что при отсутствии в комплексе ППТ все инструкции плавающей арифметики могут выполняться на аппаратуре центрального процессора путем запуска соответствующих микропрограмм, однако скорость исполнения этих инструкций будет в 5—7 раз меньше. Наличие или отсутствие ППТ определяется в процессе первоначальной загрузки микропрограмм в управляющую память центрального процессора, и в зависимости от этого осуществляется загрузка тех или иных микропрограмм. При отсутствии ППТ загружаются микропрограммы интерпретации набора инструкций плавающей арифметики, а при наличии — микропрограммы связи с аппаратурой ППТ.

Таким образом, пользовательская программа будет выполняться в любом случае, только при использовании ППТ она выполнится в несколько раз быстрее.

Когда ППТ установлен в вычислительном комплексе, то инструкции плавающей арифметики выполняются не центральным процессором, а пересылаются в ППТ по специальной 8-разрядной шине. При этом данные передаются из центрального процессора в ППТ по 32-разрядной шине данных. Получив инструкцию

с операндами, ППТ выполняет ее на своем оборудовании, используя свое собственное микропрограммное управление. Получив результат, он отсылает его назад в центральный процессор.

Блоки элементов ОП связаны с БЭ КП. Каждая плата ОП имеет емкость 1 Мбайт. Длина ячейки ОП составляет 39 разрядов, из которых 7 являются контрольными. Контроль производится по коду Хэмминга, что позволяет исправлять одиночные и обнаруживать двойные ошибки. Максимальный объем БЭ ОП 5 Мбайт.

Многофункциональный контроллер связи (МКС) подключается к ОШ и содержит аппаратуру управления для восьми последовательных асинхронных линий, одной синхронной последовательной линии и одного параллельного интерфейса. Параллельный интерфейс может использоваться либо для управления широкой печатью, либо как интерфейсный модуль общего назначения. Этот интерфейс так же, как и синхронный, ведет передачу данных в режиме прямого доступа.

Основными обрабатывающими элементами в МКС являются микропроцессорные секции К1804ВС1. Используются также микросхемы ПМЛ. Управление многофункциональным контроллером связи — микропрограммное с помощью своей управляющей памяти и своего микросеквенсера. Управляющая память реализована на микросхемах ППЗУ.

7.3. ОРГАНИЗАЦИЯ ДИСПЕТЧЕРА ПАМЯТИ И ЕГО РЕАЛИЗАЦИЯ

Как уже отмечалось в гл. 5, диспетчер памяти реализован с помощью технических и программных средств и предназначен для распределения физической ОП и управления обращением к ней. Для мультипрограммных систем в физической памяти обычно размещаются несколько процессов в одно и то же время. Чтобы гарантировать целостность каждого процесса и исключить его воздействие на другие процессы и операционную систему, в СМ1700 используется защита памяти и разделение адресного пространства на несколько областей.

Чтобы еще более увеличить надежность программных систем, для управления доступом к памяти используются четыре иерархических режима, которые по степени привилегированности распределяются следующим образом: режим ядра, режим управления, режим супервизора и режим пользователя. Наиболее привилегированным режимом является первый. Защита памяти указывается на уровне данной страницы, и при этом страница может быть недоступна, доступна только для чтения или доступна и для чтения, и для записи. Доступность страницы может указываться для каждого из четырех режимов доступа. Любая ячейка, доступная для данного режима работы, доступна также для более привилегированных режимов.

При исполнении программы центральный процессор генерирует виртуальные адреса. Однако перед тем, как эти адреса могут быть использованы для обращения к физической памяти за инструкциями и данными, они должны быть преобразованы в физические адреса. Преобразование в физические адреса осуществляется с помощью буфера трансляции, содержащего регистры преобразования и описания страниц (т. е. записи для таблиц страниц, пользуясь терминами гл. 5) и находящегося в БЭ контроллера памяти. Заполнение буфера информацией находится в компетенции операционной системы, и его содержимое определит, где каждая из виртуальных страниц длиной 512 байт будет отображена в физической памяти.

Таким образом, диспетчер памяти обеспечивает как защиту памяти, так и механизм распределения ее.

Виртуальный адрес содержит 32 разряда и определяет ячейку длиной в байт в виртуальном адресном пространстве. Для программиста этот адрес обеспечивает адресацию линейной матрицы объемом в 4 294 967 296 байт.

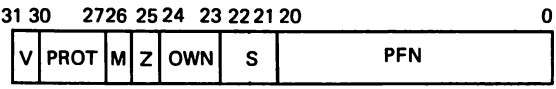
Как уже указывалось, виртуальное адресное пространство разделено на части, каждая из которых содержит 512 байт. Эти части называются страницами. Страница является объектом преобразования (отображения) и защиты. Виртуальное адресное пространство является слишком большим, чтобы быть представленным в настоящее время в физической памяти. Поэтому диспетчер обеспечивает механизм представления активной части виртуального адресного пространства в физической ОП. Кроме того, диспетчер выполняет защиту страниц между процессами. Операционная система управляет таблицами преобразования виртуальных адресов в физические, а также сохраняет неактивные, но используемые в данной программе области виртуального адресного пространства на внешних запоминающих устройствах, например на дисковых накопителях.

Трансляция или преобразование виртуального адреса в физический осуществляется установкой разряда включения преобразования памяти MME в регистре MAPEN, являющемся внутренним регистром процессора. Когда MME равен единице, диспетчер памяти включен, а когда нулю, то выключен. Во время процедуры инициализации системы MAPEN устанавливается в нуль.

Установка MME в нуль отключает трансляцию адресов и защиту памяти. Разряд n виртуального адреса VA ⟨n⟩ прямо копируется в разряд n физического адреса PA ⟨n⟩ для n от 0 до 29. Разряды VA ⟨31:30⟩ игнорируются, а PA ⟨31:30⟩ всегда устанавливаются нулями. Разряд VA ⟨n⟩ игнорируется, если PA ⟨n⟩ не существует. Защита памяти не производится, и все обращения разрешены во всех режимах.

При MME, равным единице, включается трансляция адресов и защита памяти. Для того чтобы определить, разрешено ли

Рис. 7.2. Регистр описания страницы PTE



данное обращение к памяти, процессор руководствуется следующими условиями и параметрами:

виртуальным адресом, который используется как индекс (смещение) в таблице страниц (представляющей собой набор регистров описания страницы в буфере трансляции);

типом обращения к памяти (чтение или запись);

текущим уровнем привилегированности из регистра двойного слова (в дальнейшем просто регистра) состояния процессора.

Если данный тип обращения разрешен и адрес может быть преобразован, то в результате получается физический адрес, соответствующий данному виртуальному адресу.

Указанным обращением является READ, если должна быть выполнена операция чтения, или WRITE, если должна быть выполнена операция записи. Если задана операция модификации (т. е. операция чтения, за которой последует запись), то обращение квалифицируется как WRITE.

Для трансляции виртуальных адресов в физические процессор использует регистр описания страницы PTE (или элемент таблицы страниц, или запись в таблицу страниц). На рис. 7.2 показан формат этого регистра.

Разряд достоверности (или действительности) V ⟨31⟩ определяет достоверность разряда M и поля PFN. Если V=1, то указанные разряды достоверны. Если V=0, то недостоверны.

Поле защиты PROT ⟨30:27⟩ всегда действительно и используется аппаратурой процессора, даже если V=0.

Разряд модификации M ⟨26⟩ не используется аппаратурой центрального процессора, когда разряд достоверности сброшен. Когда разряд достоверности установлен, разряд M указывает, модифицировалась ли данная страница, т. е. была ли хотя бы одна запись в эту страницу. Если M сброшен, то страница не модифицировалась.

Разряд M сбрасывается только программой. Устанавливается он аппаратурой центрального процессора, если в данной странице производилась операция записи или модификации.

Разряды OWN ⟨24:23⟩ являются резервными.

Физический номер страницы PFN ⟨20:0⟩ представляет собой старшие 21 разряд физического адреса начала (основания) физической страницы и используется аппаратурой диспетчера, только если V=1.

Разряды Z ⟨25⟩ и S ⟨22:21⟩ являются резервными.

В физический адрес памяти транслируется или виртуальный адрес, поступающий из ЦП, или адрес, поступающий из ОШ. Адрес, подлежащий трансляции, принимается в регистр виртуального адреса. Самым младшим разрядом адреса является

разряд ⟨2⟩, так как адресуется двойное слово. Самые младшие два разряда виртуального адреса, указывающие байт в двойном слове, в трансляции не участвуют и поступают только на ротатор КП. Разряды ⟨8:2⟩ виртуального адреса в трансляции также не участвуют и определяют номер (относительный адрес) ячейки в физической странице, т. е. используются как смещение или индекс по отношению к номеру (начальному адресу) физической страницы.

Буфер трансляции содержит 1К ячеек, которые разделены на три части: пространство центрального процессора (128 ячеек), которое, в свою очередь, делится на пространство системы и пространство процесса, пространство ОШ (512 ячеек) и неиспользуемое пространство (384 ячейки).

Каждая ячейка пространств процессора и ОШ, называемая регистром описания страницы или регистром преобразования, содержит 15-разрядный номер физической страницы (в СМ1700 разряды ⟨20:15⟩ поля PFN не используются) и разряд достоверности (ячейка процессора содержит еще несколько разрядов, определяющих защиту памяти,—см. выше формат PTE). Кроме того, каждая ячейка процессора имеет в буфере трансляции 16-разрядный признак, так называемый тег, который образует память тегов. Тег хранится по тому же адресу, что и связанная с ним ячейка в буфере трансляции. Тег представляет собой значение разрядов ⟨30:15⟩ виртуального адреса, используемого при обращении к памяти. Разряды ⟨14:9⟩ и ⟨31⟩ виртуального адреса определяют ячейку процессора в буфере трансляции.

Когда осуществляется обращение к физической ОП из центрального процессора, тег из соответствующей ячейки посылается на компаратор, где сравнивается с разрядами ⟨30:15⟩ виртуального адреса обращения. Если происходит совпадение, то обращение разрешено, если нет, то формируется сигнал «промаха», который вызывает микропрерывание и обращение к ЦП.

На рис. 7.3 показана для системной области схема формирования физического адреса из виртуального, поступающего из процессора.

7.4. РЕАЛИЗАЦИЯ ВВОДА-ВЫВОДА

Как отмечалось в 7.2, в качестве системного интерфейса в вычислительном комплексе СМ1700 используется ОШ, т. е. тот же интерфейс, что и в 16-разрядных мини-ЭВМ типа СМ4. В этом заключается преемственность по периферийным устройствам между этими двумя классами машин.

К ОШ в СМ1700 подключаются все периферийные устройства, кроме системных. К системным внешним устройствам относятся кассетный магнитофон и консольный терминал, подключаемые к консольной подсистеме, и дисковые накопители, подключаемые к интегрированному контроллеру диска (см. 7.2). Все остальные

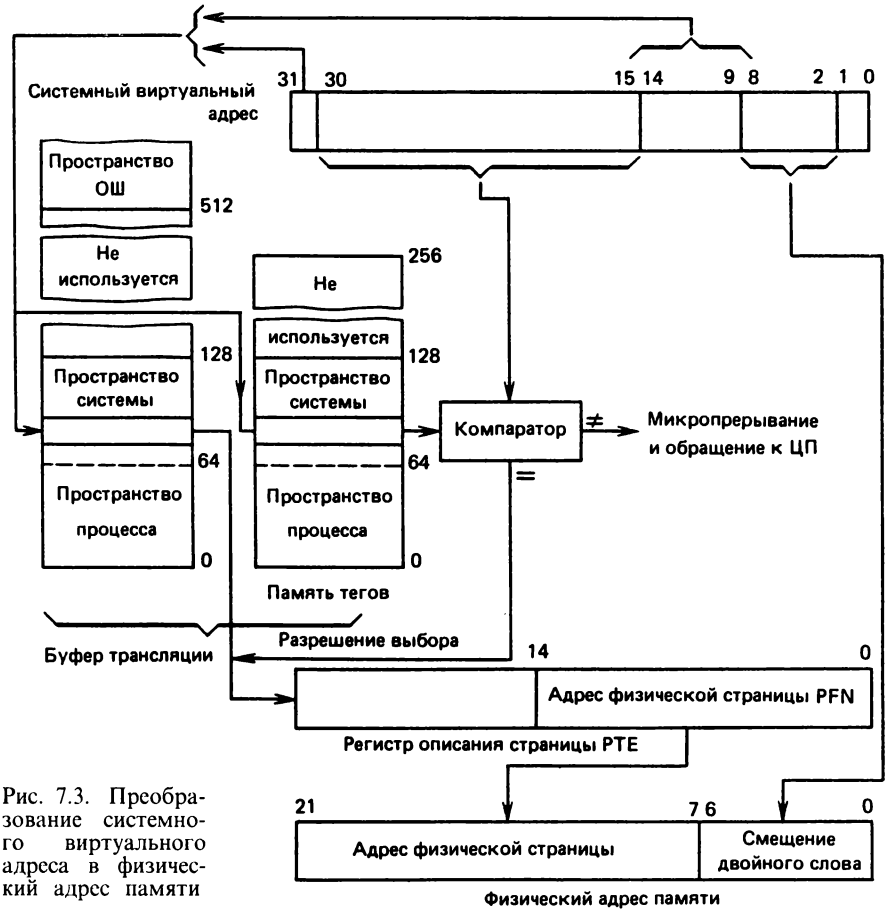


Рис. 7.3. Преобразование системного виртуального адреса в физический адрес памяти

периферийные устройства подключаются либо к ОШ, либо к МКС (см. рис. 1.1). Это не означает, что помимо системных к комплексу не могут подключаться дополнительные магнитные диски, ленты и печатающие терминалы, но все они в этом случае будут работать через ОШ, алгоритм функционирования которой хорошо известен [6].

К ОШ подключается и МКС, обеспечивающий выход на восемь асинхронных каналов связи (стык С2), один синхронный и один параллельный интерфейс ИРПР для подключения устройства параллельной печати или для использования в качестве интерфейсного модуля общего назначения. Поэтому все внешние устройства, имеющие выходы на перечисленные интерфейсы, могут подключаться к МКС. При необходимости расширения подсоединяемых периферийных устройств конструкция вычислительного комплекса СМ1700 обеспечивает подключение блока расширения системы, в который могут вставляться дополнительные МКС.

Указанное выше подключение системных и внешних устройств объясняется необходимостью разгрузить системный интерфейс ОШ, тем самым повысив производительность комплекса в целом, а также из соображений организации более эффективной диагностики.

Работу на ОШ обеспечивают специальные схемы процессора СМ1700. В них входят схемы управления ОШ, арбитр шины, регистры данных и адреса, приемопередатчики и др. В дальнейшем все эти схемы плюс схемы преобразования адресов ОШ будут называться адаптером ОШ. Адаптер ОШ в основном расположен на плате контроллера памяти и частично на плате консольно-диагностического процессора (линии данных ОШ).

Выше были кратко рассмотрены технические средства ввода-вывода. Управлением же этих средств занимаются специальные программы, имеющиеся в операционной системе. Они называются драйверами ввода-вывода.

Адаптер ОШ выполняет следующие функции:

при возникновении запросов на прерывание со стороны устройств ввода-вывода (УВВ) выделяет наиболее приоритетное;

передает запрос на прерывание от УВВ к процессору;

позволяет драйверам ввода-вывода получить доступ к регистрам внешних устройств (подключенных к ОШ), используя системные виртуальные адреса;

преобразует 18-разрядные адреса интерфейса ОШ в физические адреса оперативной памяти;

обеспечивает передачу данных в различные страницы физической памяти;

формирует адреса в границах байта.

Драйверы ввода-вывода используют адаптер ОШ для следующих целей:

чтения и записи содержимого регистров внешних устройств;

преобразования адресов ОШ в физические адреса памяти и для выполнения обмена данными по прямому доступу в память.

Драйверам тех устройств, которые не выполняют передачи данных по прямому доступу, не обязательно знать о наличии адаптера ОШ. В этом случае адаптер обеспечивает доступ к регистрам внешних устройств, используя схему преобразования адресов, которая «невидима» для драйвера, т. е. передача осуществляется между УВВ и процессором, а затем процессором и памятью и наоборот.

Каждый контроллер ввода-вывода, подключенный к ОШ, имеет набор регистров управления и состояния, а также регистров данных. Эти регистры имеют адреса из области физического адресного пространства, называемого адресным пространством интерфейса ОШ. Драйверы УВВ получают информацию о состоянии этих устройств и инициализируют их путем чтения и записи информации в соответствующие регистры этих устройств.

Вообще говоря, драйвер может работать с адресами внешних устройств точно так же, как и со всеми другими виртуальными

адресами. Драйвер может читать и писать данные в регистр УВВ, как если бы этот регистр был ячейкой памяти. Адаптер ОШ выполняет фактическое преобразование виртуального адреса в адрес на интерфейс ОШ, который и соответствует данному регистру УВВ.

Адресное пространство ОШ включает 256 Кбайт памяти, из которых 8 Кбайт резервируется для регистров внешних устройств. При работе по прямому доступу УВВ читают данные из ячеек оперативной памяти и пишут в них, используя 18-разрядный адрес ОШ. Адаптер ОШ преобразует этот 18-разрядный адрес в физический адрес памяти. Это преобразование позволяет операционной системе, драйверам ввода-вывода обращаться в одно и то же физическое адресное пространство.

Адаптер имеет для адресов ОШ 512 ячеек или регистров в буфере трансляции для выполнения преобразования адресов ОШ в физические адреса памяти. Из них используется только 496. Каждый из этих 496 регистров представляет одну страницу в адресном пространстве ОШ. Содержимое этих регистров идентифицирует номера физических страниц в оперативной памяти. Таким образом, каждая страница пространства ОШ отображается в какой-то странице физической памяти.

Для обеспечения нормальной работы при выполнении передач по прямому доступу подпрограммы операционной системы МОС ВП заполняют столько регистров преобразования (предназначенных для ОШ в буфере трансляции) действительными номерами физических страниц, сколько необходимо для выполнения данного объема передач. После этого устройство, выполняющее передачи по прямому доступу, может устанавливать адрес на ОШ. Адаптер, получив этот адрес, преобразует его в физический адрес памяти, используя следующие данные:

9-разрядное поле адреса страницы ОШ (разряды с 9-го по 17-й адреса ОШ), указывающее на регистр преобразования в буфере трансляции (аналогично регистру описания страницы при трансляции адресов процессора);

поле номера физической страницы, содержащееся в данном регистре преобразования, образующее старшую часть физического адреса памяти;

разряды со 2-го по 8-й адреса ОШ, непосредственно образующие разряды с 0-го по 6-й физического адреса памяти.

Получившийся в результате физический адрес памяти указывает на ячейку длиной в двойное слово (длинное слово), к которой и производилось обращение. Затем адаптер ОШ определяет байт внутри данного двойного слова, используя два самых младших разряда адреса ОШ.

На рис. 7.4 показана процедура преобразования адреса ОШ в физический адрес памяти.

Каждый регистр преобразования адаптера содержит разряд, называемый битом достоверности (действительности) регистра

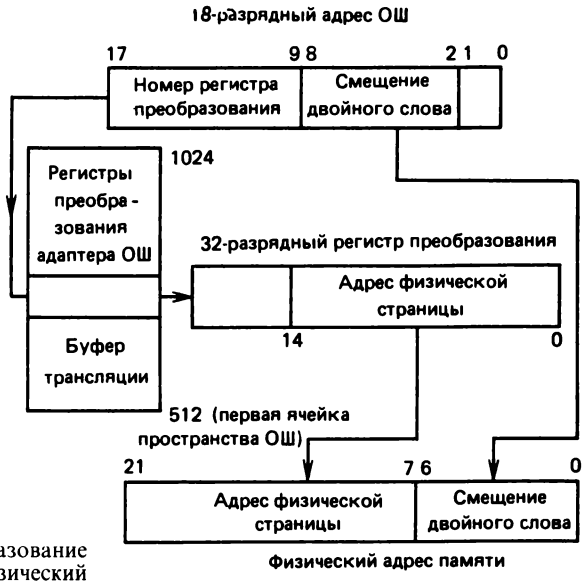


Рис. 7.4. Преобразование адреса ОШ в физический адрес памяти

преобразования. Адаптер проверяет этот бит каждый раз, когда он обращается к соответствующему регистру преобразования. Если этот бит не установлен в состояние единицы, то адаптер прекращает данное обращение. Нулевое состояние этого бита делает недостоверным содержимое данного регистра, т. е. преобразование в физический адрес запрещено, и попытка сделать это вызывает прерывание текущей работы.

Передача данных по прямому доступу в память осуществляется следующим образом:

сначала внешнее устройство устанавливает адрес на ОШ;

адаптер ОШ определяет регистр преобразования, который соответствует установленному адресу;

адаптер проверяет, установлен ли в данном регистре бит достоверности;

адаптер преобразует адрес шины в номер физической страницы;

в зависимости от типа операции, указанной на ОШ (чтение, чтение с паузой, запись или запись байта), адаптер инициирует работу контроллера памяти.

В один и тот же период времени может производиться передача с несколькими устройствами на шине. При одновременном поступлении запросов на передачу данных с этих устройств адаптер выполняет функцию арбитража. Драйвер УВВ не оказывает на выполнение этой функции никакого влияния. На один запрос прямого доступа может выполняться только одна посылка данных.

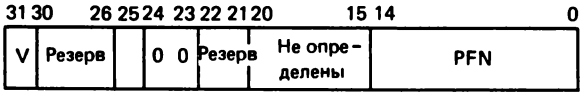


Рис. 7.5. Регистр преобразования адаптера ОШ

Последовательность запуска передач по прямому доступу, выполняемая операционной системой, следующая:

определить набор регистров преобразования;

загрузить регистры преобразования номерами физических страниц;

установить разряд достоверности в каждом регистре преобразования из выделенного набора, при этом в регистре, следующем за последним регистром преобразования в данном наборе, разряд достоверности должен быть сброшен;

загрузить начальный адрес передачи данных в соответствующий регистр внешнего устройства;

загрузить число передаваемых слов или байтом в соответствующий регистр внешнего устройства;

инициализировать передачу данных путем установки соответствующих разрядов регистра управления данного внешнего устройства.

На рис. 7.5 показан формат регистра преобразования для адаптера ОШ вычислительного комплекса СМ1700.

Разряд <31> регистра преобразования представляет собой бит достоверности. Разряды <30:26> и <22:21> являются резервными. Разряд <25> определяет смещение на байт. Разряды <24:23> должны быть равны нулю. Разряды <20:15> не определены. Разряды <14:0> содержат номер физической страницы.

Глава 8. СРАВНИТЕЛЬНЫЕ ХАРАКТЕРИСТИКИ И ВОПРОСЫ ПРИМЕНЕНИЯ СМ1700

8.1. СРАВНЕНИЕ С АРХИТЕКТУРОЙ СИСТЕМЫ КОМАНД ЕС ЭВМ

В предыдущих главах рассматривалась архитектура вычислительного комплекса СМ1700 так, как она представляется пользователю. Однако знакомство только с данной архитектурой не дает возможности полноценно ее оценить.

В этой главе рассматриваются архитектуры двух других вычислительных систем: ЕС ЭВМ и СМ ЭВМ типа СМ4. Каждая из этих систем предназначалась для различных целей, и поэтому при их создании использовались различные средства. Сравнение архитектуры СМ1700 с указанными архитектурами даст возможность лучше понять ее место среди них.

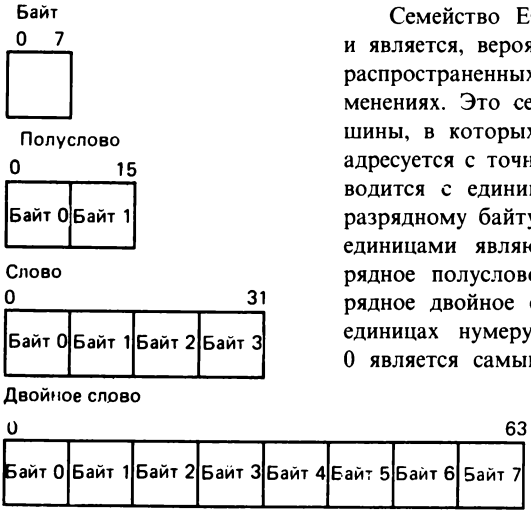


Рис. 8.1. Форматы данных семейства ЕС ЭВМ

Семейство ЕС ЭВМ появилось в 70-х годах и является, вероятно, одним из наиболее широко распространенных в самых разнообразных применениях. Это семейство представляет собой машины, в которых информация, как и в СМ1700, адресуется с точностью до байта и работа производится с единицами информации, кратными 8-разрядному байту. Основными информационными единицами являются: 8-разрядный байт, 16-разрядное полуслово, 32-разрядное слово и 64-разрядное двойное слово (рис. 8.1). Разряды в этих единицах нумеруются слева направо, и разряд 0 является самым старшим значащим разрядом.

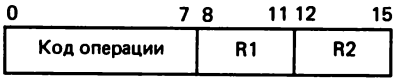
Вычислительная система ЕС ЭВМ имеет шестнадцать 32-разрядных общих регистров, пронумерованных от 0 до 15. Два подряд расположенных регистра (из которых первый четный, а второй нечетный) могут использоваться в качестве регистра двойного слова. Например, 2-й и 3-й регистры могут хранить двойное слово, а 3-й и 4-й — нет. Кроме того, имеются четыре 64-разрядных регистра для операций над числами с плавающей точкой. Они имеют номера 0, 2, 4 и 6. Соседние регистры для чисел с плавающей точкой (0,2 и 4,6) могут использоваться для хранения расширенных операндов.

Хотя машины ЕС ЭВМ являются 32-разрядными (поскольку обрабатываются главным образом 32-разрядные числа), программные адреса оперативной памяти имеют 24 разряда. Когда какой-либо регистр используется для адресации виртуальной памяти, то старшие 8 разрядов (с 0 по 7) игнорируются при вычислении адреса. Поэтому программа может адресовать 2^{24} или 16 777 216 байт памяти.

Вычисление адреса в ЕС ЭВМ проще, чем в СМ1700, так как меньше вариантов вычислений. Адрес памяти обычно формируется как сумма содержимого общего регистра (известного как базовый регистр B) и 12-разрядного целого числа (называемого смещением D). При вычислении используются только младшие 24 разряда. Это символически обозначается D(B) и похоже на адресацию смещения в СМ1700.

Второй вариант указания адреса основан на использовании индексного регистра X в дополнение к базе и смещению. Индексный регистр может использоваться как адрес элемента массива, который указывается базовым регистром. Смещение складывается с содержимым младших 24 разрядов базового и индексного регистров, и в результате получается адрес операнда. Этот режим адресации обозначается D(X, B), и он подобен режиму, применяемому в СМ1700, за исключением того, что индексный регистр не умножается на размер элемента массива. Для обращения к 32-разрядным словам массива индексный регистр должен инкрементироваться на 4 после каждого шага.

Рис. 8.2. Формат регистр-регистр



Эти два варианта являются единственными для указания адресов операндов в инструкциях системы ЕС ЭВМ. Таким образом, чтобы обратиться к какой-либо ячейке памяти, всегда должен использоваться базовый регистр. Так как смещение равно 12 разрядам, базовый регистр может адресовать блок памяти объемом 4096 байт. Следовательно, для каждого сегмента размером в 4096 байт связанного массива данных должен выделяться один из 16 общих регистров. Однако если к различным сегментам нет одновременного обращения, то можно обойтись одним базовым регистром, каждый раз загружая его адресом того сегмента, к которому в данный момент будет осуществляться обращение.

В СМ1700 код операции в инструкции указывает размер, тип и число операндов, а спецификатор операндов содержит один байт для указания используемого режима адресации. В ЕС ЭВМ код операции помимо размера и типа операндов включает и режимы их адресации. Это сокращает размер поля спецификаторов операндов, поскольку режимы адресации задаются в неявной форме. При таком подходе может потребоваться большое число кодов операции при необходимости использовать разнообразное количество типов адресации, но, как уже указывалось, в ЕС ЭВМ применяется ограниченное число режимов адресации с целью сохранить эффективность кодировки инструкции.

Длина инструкций в ЕС ЭВМ может быть 2, 4 или 6 байт. Имеется пять форматов инструкций для реализации различных способов указания операндов источников и приемников. Рассмотрим кратко каждый из них.

1. Формат регистр-регистр RR.

В формате регистр-регистр оба операнда расположены в регистрах (рис. 8.2).

Ассемблерный синтаксис для формата RR:

КОД ОПЕРАЦИИ R1,R2

Например, инструкция SR вычитает из содержимого R1 содержимое R2 и помещает разность на место первого операнда:

SR 1,2

Заметим, что в СМ1700 результат был бы записан во второй регистр.

2. Формат регистр-память с индексацией RX.

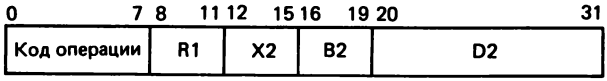
В этом формате один операнд размещается в регистре, а второй — в ячейке памяти, которая задается с помощью базы, индекса и смещения (рис. 8.3).

Ассемблерный синтаксис для формата RX:

КОД ОПЕРАЦИИ R1,D2(X2,B2)

Код операции определяет, кто является источником, а кто приемником.

Рис. 8.3. Формат регистр с индексацией



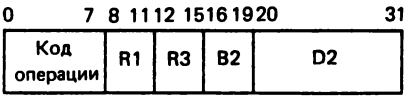


Рис. 8.4. Формат регистр-память

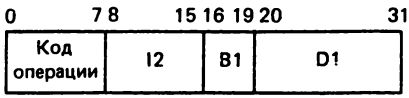


Рис. 8.5. Формат с непосредственным операндом

Например, инструкция A (сложение)

A R,D(X,B)

складывает содержимое ячейки памяти и регистра R, а результат запоминает в регистре. Инструкция ST (запись в память)

ST R,D(X, B)

записывает содержимое регистра в указанную ячейку памяти.

3. Формат регистр-память без индексации RS.

Этот формат определяет три операнда: два регистра и одну ячейку памяти, обращение к которой осуществляется через базовый регистр и смещение (рис. 8.4).

Ассемблерный синтаксис для инструкций RS:

КОД ОПЕРАЦИИ R1,R3,D2(B2)

Например, инструкция LM (загрузка групповая)

LM R1,R3,D(B)

загружает общие регистры, начиная с регистра, заданного первым операндом, в ряд, кончая регистром, заданным полем R3, из последовательно расположенных ячеек памяти с начальным адресом, определяемым суммой содержимого базового регистра и смещения.

4. Формат с непосредственным операндом в инструкции (SI).

В этом формате, показанном на рис. 8.5, 8-разрядный непосредственный операнд и ячейка памяти указываются в инструкции.

Ассемблерный синтаксис для формата SI:

КОД ОПЕРАЦИИ D1(B1),I2

Например, инструкция пересылки (MVI)

MVI 11(5),7

пересылает непосредственное значение 7 в ячейку памяти, которая определяется суммой содержимого базового регистра 5 и смещения 11.

5. Формат память-память (SS).

Формат память-память используется только для работы со строками. Адрес каждого операнда определяется базовым регистром и смещением. Имеется два формата память-память. В одном из них для каждого операнда есть 4-разрядное поле, в котором указывается длина операнда, а в другом для обоих операндов — только одно 8-разрядное поле. Машинный формат для инструкций SS показан на рис. 8.6, *а*, *б*.

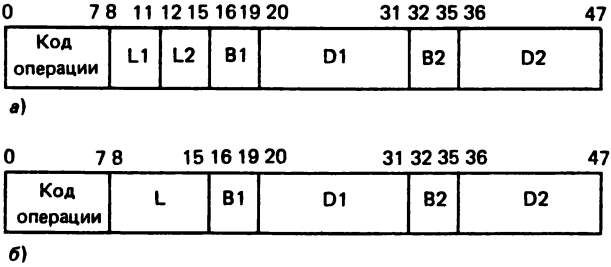
Ассемблерный синтаксис для инструкций SS:

КОД ОПЕРАЦИИ D1(L1, B1),D2(L2,B2)

или

КОД ОПЕРАЦИИ D1(L,B1),D2(B2)

Рис. 8.6. Формат память-память



Например, инструкция пересылки строки символов

```
MVC  0(32,2),8(3)
```

пересылает строку символов длиной 32 байта из ячейки, определяемой базовым регистром 3 и смещением 8, в ячейку, определяемую базовым регистром 2 и смещением 0.

Система ЕС ЭВМ имеет арифметику, которая работает в форматах регистр-регистр и память-регистр, но не может работать в форматах регистр-память и память-память. Поэтому промежуточный регистр должен использоваться для сложения содержимого двух ячеек памяти. Чтобы сложить ячейки памяти X и Y, необходимо выполнить следующие инструкции:

```
L  R1,X    ;загрузить X в R1
A  R1,Y    ;сложить Y и R1
ST R1,Y    ;запомнить результат в Y
```

Все три инструкции представляют собой инструкции формата RX. Тем не менее, заметим, что место операнда-источника и операнда-приемника указывается с помощью кода операции. Для обозначения символьных операндов, таких как X и Y, программист сообщает ассемблеру, какой регистр должен быть базовым, а затем ассемблер вычисляет смещение.

В системе команд ЕС ЭВМ в отличие от СМ1700 не все имеющиеся единицы информации могут использоваться в инструкциях. Например, имеется только 5 инструкций для сложения целых чисел:

1. Сложение двух 32-разрядных чисел со знаком, расположенных в регистрах (AR).
2. Сложение двух 32-разрядных чисел со знаком, находящихся в ячейке памяти и регистре (A).
3. Сложение 16-разрядного числа со знаком из ячейки памяти с 32-разрядным числом со знаком, находящимся в регистре (AH для сложения полуслова).
4. Сложение двух 32-разрядных чисел без знака из общих регистров (AL для логического сложения).
5. Сложение 32-разрядного числа без знака из ячейки памяти с аналогичным числом из регистра (ALR).

После такого краткого анализа архитектуры системы команд ЕС ЭВМ можно сделать следующие выводы:

1. Архитектура системы команд ЕС ЭВМ имеет менее общий характер в формировании инструкций, чем у СМ1700, но эквивалентные режимы адресации представлены более компактно.

2. Адресация памяти в ЕС ЭВМ требует базового регистра, с помощью которого можно адресовать блок длиной в 4096 байт. Так как полный 24-разрядный адрес не может быть прямо представлен в инструкции, то произвольная адресация больших массивов не так удобна, как в системе СМ1700.

3. Система ЕС ЭВМ, как и СМ1700, имеет индексные регистры, однако их содержимое не модифицируется автоматически на размер элемента массива, к которому было совершено обращение. Поэтому в ЕС ЭВМ имеются команды, инкрементирующие индексные регистры и проверяющие их верхние границы в целях облегчения работы с ними.

4. Общие регистры ЕС ЭВМ не могут использоваться для операций над числами с плавающей точкой. Для представления расширенных операндов может использоваться только пара из четного и нечетного регистров. Кроме того, не реализованы режимы автоинкрементирования и автодекрементирования.

5. Система ЕС ЭВМ не имеет арифметики, работающей в формате память-память, поскольку при разработке архитектуры внимание было направлено на то, чтобы уменьшить размер инструкции и упростить ее кодирование. В результате инструкции ЕС ЭВМ достаточно короткие, от двух до шести байт, в то время как инструкции СМ1700 могут быть очень длинными. Достаточно трудно сравнивать эти две архитектуры, поскольку несколько коротких инструкций ЕС ЭВМ могут интерпретироваться одной длинной инструкцией СМ1700.

6. Для сохранения компактности представления системы команд в ЕС ЭВМ нет также косвенных режимов адресации.

8.2. СРАВНЕНИЕ С АРХИТЕКТУРОЙ 16-РАЗРЯДНЫХ МИНИ-ЭВМ ТИПА СМ4

Семейство 16-разрядных мини-ЭВМ типа СМ4 (СМ3, СМ4, СМ1300, СМ1420, СМ1600) является предшественником 32-разрядной мини-ЭВМ СМ1700. Как видно из материалов этой книги, СМ1700 разработана таким образом, чтобы по возможности максимально сохранить общие черты с мини-ЭВМ типа СМ4. К ним относятся:

организация ввода-вывода и системный интерфейс (ОШ);

некоторые форматы представления данных и режимы адресации;

использование сходного синтаксиса в ассемблерном языке.

К этому следует добавить, что в СМ1700 имеется режим совместимости для выполнения непривилегированных инструкций машин типа СМ4[1] (за исключением инструкций плавающей арифметики). Этот режим позволяет исполнение пользовательских программ, написанных для СМ4, под управлением ОС РВ.

[1] Далее просто СМ4, если специально не оговаривается конкретная модель.

Однако между 16-разрядными мини-ЭВМ и 32-разрядной СМ1700 имеются существенные различия. К ним в первую очередь относится расширение виртуального адресного пространства в СМ1700. Если машины первого типа используют 16-разрядные адреса, обеспечивая возможность адресации 2^{16} или 65 536 ячеек памяти (что часто недостаточно для решения сложных задач и обработки больших структур данных), то в архитектуре СМ1700 заложен 32-разрядный адрес, позволяющий увеличить адресное пространство до 2^{32} или 4 294 967 296 различных ячеек.

Значительное увеличение объема виртуальной памяти дает возможность по-новому подходить как к разработке прикладных программ, так и к организации и построению операционных систем.

К другим отличиям между этими классами машин можно отнести:

значительное увеличение у СМ1700 числа машинных инструкций, отсутствующих в системах команд СМ4 (инструкции десятичной арифметики и обработки символьных строк, инструкции обработки битовых полей и работы с очередями, инструкции, ориентированные на эффективную реализацию операционных систем, и др.);

увеличение у СМ1700 числа обрабатываемых форматов данных;

увеличение производительности за счет обработки информации в 32-разрядной сетке;

использование более совершенной элементной базы, основу которой составляет программируемая матричная логика, позволяющая сократить затраты на аппаратуру в 4 раза;

наличие мощного диагностического обеспечения, включающего микродиагностику (в форматах микрокоманд) и макродиагностику (в форматах машинных инструкций).

Далее рассмотрим кратко основные архитектурные особенности СМ4, сопоставляя их с архитектурными особенностями СМ1700.

Основными единицами обрабатываемой информации в СМ4 являются 8-разрядные байты и 16-разрядные слова. Для каждого из этих форматов имеются свои инструкции. Некоторые типы инструкций работают с 32-разрядными двойными словами. Нумерация разрядов производится справа налево: в слове — от 0 до 15, в байте — от 0 до 7. Самым младшим разрядом является нулевой.

Адресация памяти осуществляется с точностью до байта. Два рядом расположенных байта образуют слово, из которых младший байт имеет четный адрес, а старший — нечетный (рис. 8.7). Слово адресуется так же, как входящий в него младший байт, т. е. оно всегда имеет четный адрес.

Напомним, что в СМ1700 инструкции могут работать с 8-разрядными байтами, 16-разрядными словами, 32-разрядными двойными словами и 64-разрядными учетверенными словами.

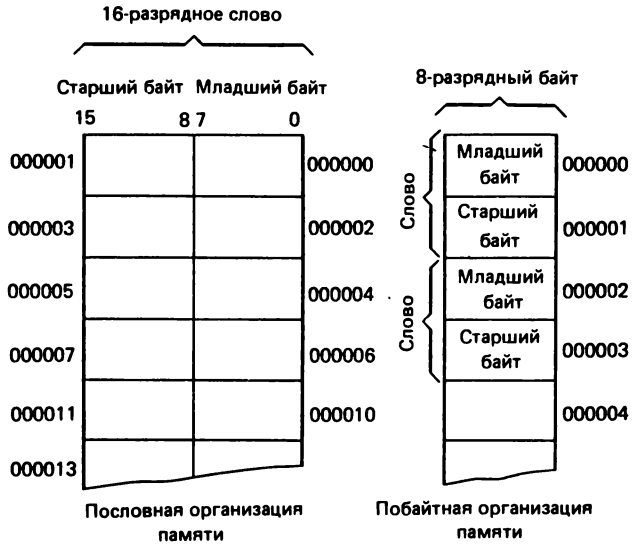


Рис. 8.7. Адресация памяти в ЭВМ типа СМ4

Каждая из указанных информационных единиц сформирована из групп последовательно расположенных байт и всегда адресуется так же, как самый младший байт в данной группе.

Форматы данных, обрабатываемые машинными инструкциями СМ4, делятся на три группы: логические коды, числа с фиксированной точкой (целые числа) и числа с плавающей точкой. Логическими кодами могут быть представлены символьные величины, числа без знака и битовые величины.

Инструкции, обрабатывающие числа с плавающей точкой, различны для разных моделей 16-разрядных ЭВМ, а в некоторых моделях (СМ3) они отсутствуют вообще. Модель СМ4 имеет 4 инструкции плавающей арифметики и работает с операндами только одинарной точности (32 разряда). Модели СМ1420 и СМ1600 содержат более 40 инструкций плавающей арифметики для работы с операндами одинарной и двойной (64 разряда) точности, включая инструкции преобразования данных из одних форматов в другие.

В СМ1700 помимо указанных форматов данных имеются битовые поля переменной длины, десятичные числа и символьные строки. Что касается арифметики с плавающей точкой в СМ1700, то для ее реализации используется около 90 инструкций и помимо операндов одинарной и двойной точности обрабатываются операнды форматов G — с расширенным порядком (64 разряда) и H — учетверенной точности с расширенным порядком (128 разрядов).

В СМ4 содержится 8 общих 16-разрядных регистров, пронумерованных от 0 до 7 (в СМ1700 имеется 16 общих 32-разрядных регистров). В зависимости

Рис. 8.8. Формат первого слова двухадресной инструкции

от режима адресации регистр, указываемый в машинной инструкции, содержит непосредственно операнд либо адрес операнда, либо индекс.

Система команд включает в себя одно- и двухадресные инструкции, инструкции передачи управления (инструкции прерываний, перехода и инструкции для работы с подпрограммами), служебные инструкции. На рис. 8.8 представлен формат первого слова двухадресной инструкции (в зависимости от режима адресации каждого из операндов двухадресные инструкции могут состоять из одного, двух или трех слов). Первое слово инструкции всегда содержит код операции и спецификаторы двух операндов. Результат двухадресной инструкции записывается на место второго операнда — операнда приемника.

Напомним, что в СМ1700 число операндов, указываемых в одной инструкции, может достигать шести.

Режимы адресации операндов в машинных инструкциях СМ4 задаются специальными кодами, местоположение которых в инструкциях строго фиксировано. Имеются следующие виды адресации: прямая; косвенная (ступенчатая); с использованием РС. Каждому из указанных видов соответствует по 4 режима адресации.

Режимы прямой адресации включают:

адресацию через регистр, когда операнд задается непосредственно в регистре;

адресацию с автоувеличением, когда регистр содержит адрес операнда и после выполнения инструкции его содержимое увеличивается на 2 или 1;

адресацию с автоуменьшением, при котором перед выполнением операции содержимое регистра с адресом операнда уменьшается на 2 или 1;

адресацию с индексацией, при которой адрес равен сумме содержимого регистра и смещения, которое является вторым или третьим словом инструкции.

Режимы косвенной адресации включают:

адресацию через регистр, когда регистр содержит адрес операнда;

адресацию с автоувеличением, когда регистр содержит адрес адреса операнда и после выполнения операции его содержимое увеличивается на 2;

адресацию с автоуменьшением, когда перед операцией содержимое регистра уменьшается на 2 и рассматривается как адрес адреса операнда;

адресацию с индексацией, при которой адрес адреса операнда определяется суммой содержимого регистра и смещения.

Режимы адресации с использованием РС включают в себя:

непосредственную адресацию, при которой операнд задается во втором или третьем слове инструкции;

абсолютную адресацию, при которой адрес операнда задается во втором или третьем слове инструкции;

относительную адресацию, при которой адрес операнда является суммой смещения, размещаемого во втором или третьем слове инструкции, и содержимого PC после считывания этого смещения;

относительно-косвенную адресацию, в которой задается адрес адреса операнда аналогично предыдущему режиму.

В двухадресной инструкции каждый из двух операндов может быть задан с использованием любого из рассмотренных выше режимов адресации. Однако существует ряд ограничений, связанных с ограниченными возможностями 16-разрядного поля инструкции. Так, для двухадресных команд фиксированной арифметики ASH, ASHC, MUL и DIV не оказалось возможным задавать любой режим адресации для обоих операндов, и в результате операнд приемника может находиться только в одном из общих регистров (для указания спецификатора-приемника имеется только три разряда в теле инструкции).

Эта ограниченность проявилась и в представлении инструкций плавающей арифметики. В модели СМ4 для нее используется совершенно особый формат инструкции, состоящий из кода операции и регистра-указателя стека операндов. В моделях СМ1420 и СМ1600 для плавающей арифметики применяется традиционный формат инструкции, однако один из операндов должен находиться в специальном аккумуляторе процессора с плавающей точкой, и, следовательно, появляются специальные инструкции для загрузки этих аккумуляторов из ячеек памяти.

От всех этих ограничений удалось избавиться в СМ1700, в инструкциях которой каждый операнд может использовать любой режим адресации.

8.3. ХАРАКТЕРИСТИКИ БАЗОВЫХ КОМПЛЕКСОВ СМ1700

В настоящее время пользователи могут заказать 14 базовых одномашинных комплексов (комплектов поставки) СМ1700.01 — СМ1700.14.

Все базовые комплексы строятся на базе вычислительной машины СМ2700, которая включает процессор арифметико-логический (СМ2700.2400), процессор с плавающей точкой (СМ2700.2008), процессор консольный (СМ2700.2805), контроллер ОЗУ (СМ2700.2007) и модуль ОЗУ емкостью 1 Мбайт (СМ1700.3522), контроллер связи многофункциональный (СМ1700.4304). К консольному процессору по стыку С2-ИС подключены:

устройство печати консольное СМ6380;

устройство загрузки КНМЛ СМ5218.

Таймер, встроенный в консольный процессор, обеспечивает возможность выдачи сигналов с дискретностью $\tau = 10$ мкс.

Используя контроллеры, включенные в состав СМ2700, к вычислительной машине могут быть дополнительно подключены:

к контроллеру ОЗУ

1—4 модуля ОЗУ СМ1700.3522 (с соответствующим увеличением общего объема ОЗУ до 2 ... 5 Мбайт);

к контроллеру связи многофункциональному

устройства, выходящие на асинхронный канал по стыку С2-ИС (до восьми каналов);

устройство, выходящее на стык ИРПР (1 канал);

дополнительно еще имеется один канал межмашинной синхронной связи с другим вычислительным комплексом СМ ЭВМ по стыку С2-ИС с поддержкой протоколов DDCMP или HDLC.

Группа ВЗУ включает:

устройство запоминающее внешнее СМ1700.5309 с одним НМЛ СМ5309 (объем 20 Мбайт);

устройство внешней памяти на магнитной ленте кассетное СМ5218 (2×256 Кбайт);

устройство запоминающее внешнее СМ1700.5408 (с двумя НМД СМ5408 2×16 Мбайт) и СМ1700.5408-01 (с пятью НМД СМ5408 5×16 Мбайт);

устройство запоминающее внешнее СМ1700.5514 (с тремя НМД типа «винчестер» СМ5514 3×20 Мбайт);

устройство запоминающее внешнее типа «винчестер» СМ1700.5504 (с одним НМД СМ5504 120 Мбайт) и СМ1700.5504-01 (с двумя НМД СМ5504 2×120 Мбайт).

Как отмечалось, для расширения ОЗУ используется модуль ОЗУ СМ1700.3522 объемом 1 Мбайт.

Устройства растровой печати представлены в двух модификациях: СМ6334 и СМ6380.01 (последнее устройство предназначено для получения «твердой копии» с экрана).

Для формирования комплектов поставки используются три типа видеотерминалов из номенклатуры СМ ЭВМ:

алфавитно-цифровой СМ7238.00;

алфавитно-цифровой полиграфический СМ7238.01;

графический растровый цветной СМ7317.

Планшетные устройства полуавтоматического ввода графической информации представлены двумя модификациями: СМ6423.02, СМ6424.03 и СМ6423.

Для графического вывода используются планшетные устройства: СМ6470.02 и СМ6408.02 (с комплектом подключения СМ6408.02 к СМ1700).

В качестве дополнительного системного устройства используется комплект СМ1700.4304, включающий контроллер многофункциональный.

Указанные в табл. 8.1 комплекты, ориентированные на основные области применения при создании САПР, гибких автоматизированных производств (ГАП), систем обработки планово-экономической, статистической и учетной информации, систем автоматизации комплексных научных исследований, информационно-справочных и обучающих систем, характеризуются следующими граничными параметрами:

объем ОЗУ 1 ... 5 Мбайт;

объем ВЗУ 32 Мбайт (СМ1700.01) — 272 Мбайт (СМ1700.07 — СМ1700.10, СМ1700.13);

число рабочих мест 1 (СМ1700.01) — 12 (СМ1700.04).

Число каналов подключения внешних устройств определяется числом многофункциональных контроллеров. В комплексах СМ1700.04, .06, .07, .12, .13, .14 используются два, а во всех остальных — один многофункциональный контроллер, каждый из которых реализует 10 каналов (ИРПР, синхронный межмашинной связи и 8 асинхронных, работающих по стыку С2-ИС).

Таблица 8.1. **Комплект поставки**

Шифр изделия	Число модулей в комплектах поставок СМ1700, шт														Примечание
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	
Машина вычислительная СМ2700, в том числе:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
процессор СМ2700.2400;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
процессор с плавающей точкой СМ2700.2008;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
процессор консольный СМ2700.2805;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
контроллер ОЗУ СМ2700.2007;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
модуль ОЗУ СМ1700.3522;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
контроллер связи многофункциональный СМ1700.4304;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
заглушка ОШ;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
пульт инженерный	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
панель распределительная;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
панель распределительная;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
соединитель ОШ;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
стойка;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
устройство внешней памяти на магнитной ленте консольное СМ5218;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
устройство печати консольное СМ6380;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
стол, модификация 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Устройство запоминающее внешнее СМ1700.5309	–	–	1	–	–	1	1	1	1	1	1	1	1	1	С одним НМЛ СМ5309
Устройство запоминающее внешнее СМ1700.5408	1	–	1	–	1	–	1	1	–	1	1	–	1	–	С двумя НМД СМ5408
Устройство запоминающее внешнее СМ1700.5408.01	–	–	–	1	–	1	–	–	–	–	–	–	–	–	С памятью НМД СМ5408
Устройство запоминающее внешнее СМ1700.5514	–	1	–	–	1	1	–	–	1	–	1	1	–	1	С тремя НМД СМ5514
Устройство запоминающее внешнее СМ1700.5504	–	–	1	–	–	–	–	–	–	–	–	1	–	–	С одним НМД СМ5504
Устройство запоминающее внешнее СМ1700.5504.01	–	–	–	–	–	–	–	1	1	–	1	–	–	1	— С двумя НМД СМ5504

Окончание табл. 8.1.

Шифр изделия	Число модулей в комплектах поставок СМ1700, шт														Примечание
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	
Модуль ОЗУ СМ1700.3522	1	1	1	1	1	3	4	3	1	2	1	2	2	1	
Расширитель интерфейсов	1	—	1′	1	1	1	1	1	—	1	1	1	1	—	
Комплект СМ1700.4304	—	—	—	1	—	1	1	—	—	—	—	1	1	1	Включает: контроллер многофункциональный; панель распределительную; жгуты
Стол, модификация 1	1	2	4	13	1	4	8	5	2	4	2	5	2	2	
Стол, модификация 2	—	1	—	—	1	1	—	1	—	—	—	6	6	4	
Видеотерминал алфавитно-цифровой СМ7238.00	1	1	4	12	1	4	4	1	1	2	2	4	—	2	
Видеотерминал алфавитно-цифровой с графическими возможностями СМ7238.01	—	—	—	—	—	—	4	3	1	2	—	6	8	2	
Видеотерминал растровый графический цветной СМ7317	—	1	—	—	1	1	—	1	—	—	—	—	—	—	
Устройство внешней памяти на магнитной ленте кассетное СМ5219	—	1	—	—	—	—	—	—	—	—	—	—	—	—	
Устройство ввода графической информации планшетного типа СМ6424.03	—	1	—	1	1	1	—	1	—	—	—	—	—	1	
Устройство ввода графической информации планшетного типа СМ6423	—	—	—	—	—	—	1	1	—	—	—	—	—	—	
Устройство вывода графической информации СМ6470.02	—	1	—	—	—	—	—	1	—	—	—	1	—	1	
Устройство вывода графической информации СМП6408.02 ТУ 25.0800.0528 85	—	—	—	1	1	1	1	1	—	—	—	—	—	—	
Комплект подключения СМП6408.02 к ВК СМ1700	—	—	—	1	1	1	1	1	—	—	—	—	—	—	
Устройство печати растровое СМ6334	—	—	1	1	—	1	1	1	—	1	—	1	1	1	
Устройство печати консольное СМ6380.01	—	—	—	—	—	—	—	—	—	—	—	6	6	4	
Комплект запасных частей	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Комплект принадлежностей	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Комплект эксплуатационных документов	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Комплект системного программного обеспечения МОС ВП	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Комплект программного диагностического обеспечения	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Комплект микродиагностического обеспечения	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

На рис. 8.9 показана структурная схема комплекта поставки СМ1700.12. В силу того, что этот базовый комплекс обеспечивает

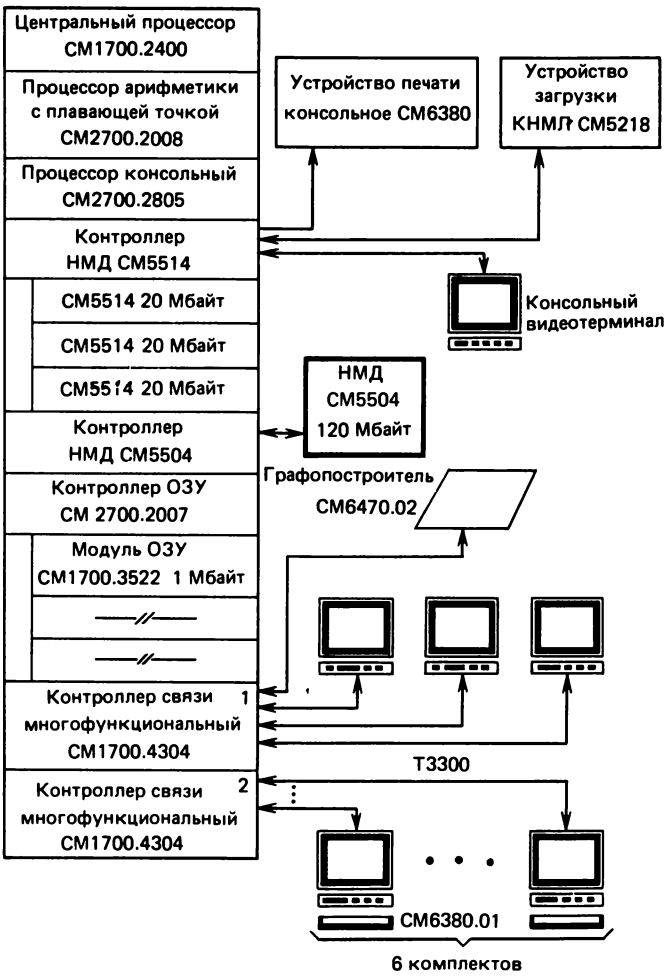


Рис. 8.9. Структурная схема комплекта поставки СМ1700.12

работу достаточно большого числа рабочих мест (10), он может быть рекомендован для применения на верхних уровнях управления в комплексных интегрированных АСУ, для реализации задач оперативного управления и диспетчеризации, построения систем групповых рабочих мест администраторов-специалистов, операторов.

8.4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ СИСТЕМОТЕХНИЧЕСКИХ ХАРАКТЕРИСТИК СМ1700

Основными задачами эффективного применения средств вычислительной техники (СВТ) на современном этапе наряду с повышением масштабов внедрения

до уровня крупносерийного и массового использования остаются разработка и поставка пользователям СВТ, удовлетворяющих требованиям комплексной автоматизации сложных производств, автоматизации проектирования, контроля и измерений, профессионального обучения и переподготовки кадров. При этом необходимо обеспечивать поставку пользователям вычислительных машин, удовлетворяющих целому комплексу разноречивых требований, важнейшими из которых остаются:

высокие технико-экономические показатели при предельно низкой стоимости, соответствующей массовому внедрению;

обеспечение надежности на уровне надежности технологического оборудования;

ускорение процесса проектирования и внедрения АСУ (близкого к «под ключ»);

эффективная работа с широким набором функциональных периферийных устройств в системах типа АСУТП, САПР, АСКИО, АСНИ, в обучающих системах и тренажерах и т. д.;

высокая реактивность, упрощение способа общения с оператором, повышенная производительность (на уровне современных ЭВМ классов супермини- и больших ЭВМ), создание распределенных систем обработки информации.

Удовлетворение всех перечисленных требований возможно только в рамках развивающегося семейства ЭВМ, включающего наборы машин от промышленных контроллеров, микроЭВМ до мини- и супермини-ЭВМ, которые объединены в семейство СМ ЭВМ, разрабатываемое в странах социалистического содружества.

Для уточнения областей применения СМ1700 наряду с табл. 8.1, где даны составы первых 14 базовых комплексов СМ ЭВМ, целесообразно рассмотреть обобщающие материалы по сравнительным характеристикам основных моделей СМ ЭВМ.

На рис. 8.10 приведены относительные показатели удовлетворения моделями СМ1700, СМ1420, СМ1300.01, СМ1800, СМ1810 требований (ОПУТ) следующих сфер применения:

I — научно-технические и инженерные расчеты;

II — экономические расчеты;

III — сетевые конфигурации и многомашинные системы для АСУТП;

IV — локальная сеть (предприятие, организация);

V — интеллектуальные периферийные устройства, однопультовый АРМ;

VI — многофункциональный терминал;

VII — обработка текстов;

VIII — работа с графикой;

IX — многофункциональное УСО;

X — встроенные применения;

XI — реальное время;

XII — обработка информации в измерительных системах.

Из приведенных материалов можно сделать вывод, что СМ1700 прежде всего ориентирована на применение в САПР (автономные графические АРМ и многомашинные системы на основе базовых комплексов СМ1700.02, СМ1700.04 — СМ1700.10); в АСУТП, САНЭ и ГАП на (2—4)-м уровнях для решения комплексных задач группового управления, подготовки производства и управления (комплексы СМ1700.01, СМ1700.03, СМ1700.04, СМ1700.07); в АСУП (обработка планово-экономической и учетно-статистической информации),

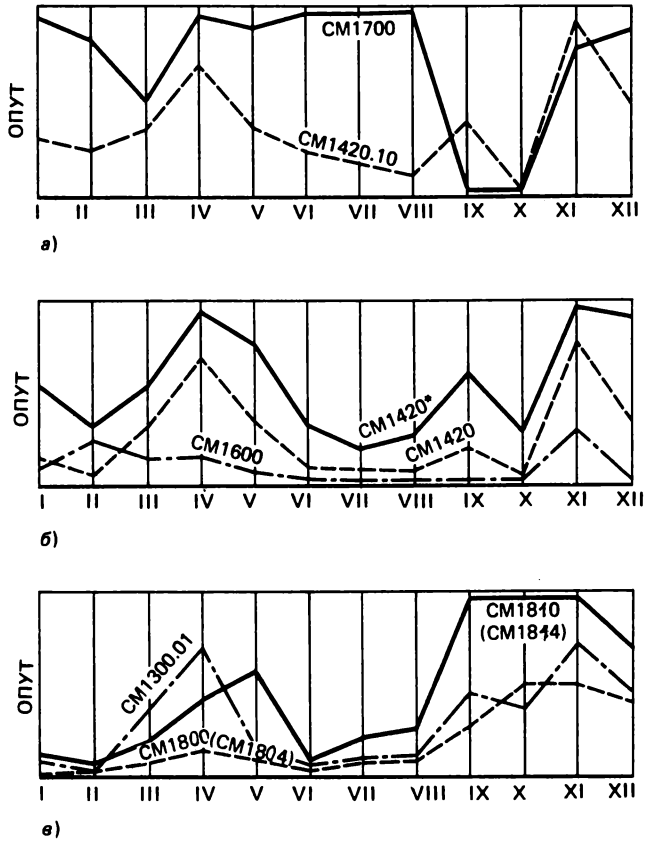


Рис. 8.10. Относительные системотехнические показатели
а — мегамини-ЭВМ; *б* — мини-ЭВМ; *в* — микроЭВМ

в информационно-справочных и обучающих системах (комплексы СМ1700.01, СМ1700.03, СМ1700.04).

Краткая классификация уровней управления в комплексных интегрированных системах управления приведена выше, а также в [3]. Там же анализируется общая структурная схема многомашинной иерархической вычислительной системы на базе СМ ЭВМ (с выходом на ЕС ЭВМ на верхнем уровне управления).

В табл. 8.2 приводятся характеристики применения СМ1700 в структуре комплексных интегрированных АСУ типа ГАП на уровнях II и III—IV. Рекомендации по возможным применениям базовых вычислительных комплексов (БВК) СМ1700 на указанных уровнях приводятся на примере основных подсистем, на которых целесообразно использование СМ1700. В этой же таблице приводятся рекомендации по применению сетевых средств связи СМ1700. Характеристики этих средств, их структурные параметры приведены на рис. 8.11.

Основные уровни комплексирования СМ1700 для класса локальных групповых систем управления иллюстрируются рис. 8.12, где рассматриваются два варианта

построения специального вычислительного комплекса программного управления высокоскоростными координатно-измерительными машинами (СВКПУ КИМ).

Высокоскоростные КИМ применяются для реализации автоматического измерения изделий сложной формы с обработкой результатов измерений и выдачей протоколов измерений. Обеспечение предельно высокой точности (микрометры) при высокой скорости измерения, возможность обработки результатов измерений по сложным алгоритмам, в том числе расчет координат, определение положения в пространстве осей и т. п., в значительной степени определяют технологический уровень современного управления. Это определяет актуальность создания систем типа КИМ.

В первом варианте комплексирования СВКПУ КИМ СМ1700 является двухмашинным. В стойку управления КИМ встроена СМ1300.01, которая обеспечивает реализацию подсистемы программного управления (ПУ) благодаря связи с модулями цифровой индикации и управления приводами КИМ через УСО СМ9104.12. Связь СМ1300.01 (ПУ) с СМ1700 осуществляется по каналу ИРПС. Обрабатывающая машина СМ1700 (ВК) строится на базе СМ2700 с расширением объема ОЗУ до 2 Мбайт (дополнительный модуль СМ1700.3522), подключением «винчестерских» НМД СМ1700.5514 (3×20 Мбайт), планшета для вывода графической информации СМ6470.02 и консольной печати (с графическими возможностями) СМ6329.

Таблица 8.2. **Применение СМ1700 в структуре комплексных интегральных АСУ типа ГАП**

Уровень управления. Характеристика подсистемы	Рекомендуемые БВК для построения комплексов							Предпочтительные сетевые средства связи
	Тип	ОЗУ [Мбайт]	ВЗУ [Мбайт]	Число рабочих мест	Графический ввод	Графический вывод	Графический дисплей	
II								
Групповое управление	СМ1700.01	2	32	1	—	—	+	СПО СЕТЬ МИНИ, СПО СЕТЬ СМ
Технологическая и конструкторская подготовка производства	СМ1700.03	2	152	4	—	—	+	СПО СЕТЬ СМ
	СМ1700.02	2	60	1	2	—	+	СПО МАГИСТР,
	СМ1700.08	4	272	5	2	2	1—4	СПО СЕТЬ
	СМ1700.06	4	140	5	1	1	1	МИНИ
III—IV								
Оперативное управление, диспетчеризация:	СМ1700.12	3	180	10	—	1	6	СПО КОЛОС
	СМ1700.13	3	272	8	—	—	8	
групповые рабочие места администратора, специалиста;	СМ1700.07	5	272	8	1	1	8	
	СМ1700.13	3	272	8		—	8	СПО МАГИСТР,
	СМ1700.12	3	180	10	—	1	6	СПО ЭМУЛЯТОР
единая база данных цеха;	СМ1700.13	5	272	8	—	1	6	СПО ТРАЛ,
	СМ1700.10	4	272	4	—	—	2	СПО ДЕМОН
САПР многопультовая	СМ1700.06	4	140	5	1	1	1	СПО МАГИСТР,
	СМ1700.08	4	272	5	2	2	1—4	СПО ТРАЛ, СПО КОЛОС

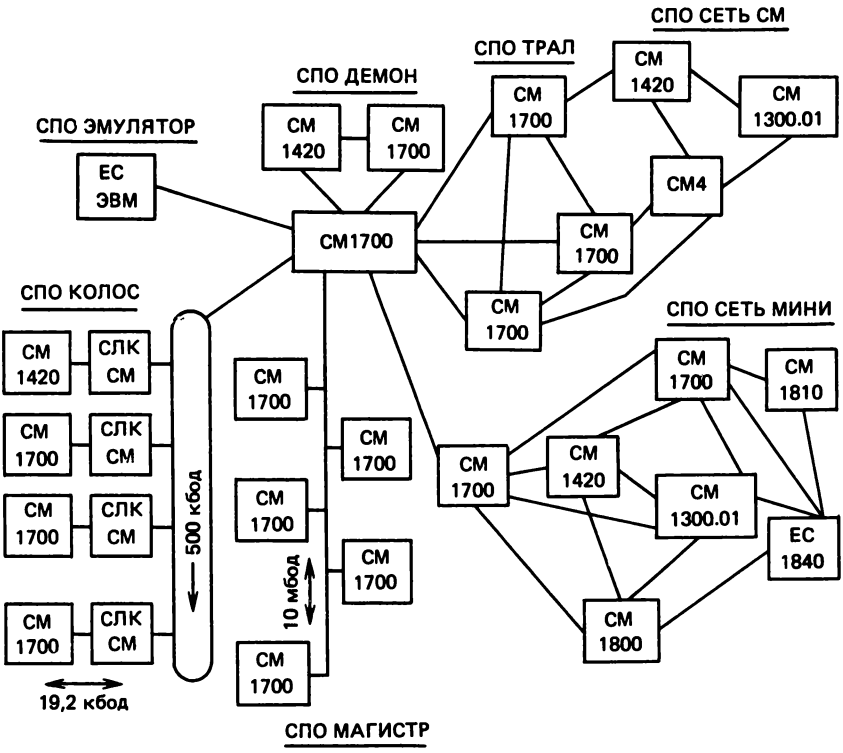


Рис. 8.11. Структура сетевых конфигураций

Во втором варианте СВКПУ КИМ СМ1700 реализуется на базе одномашинного комплекса. Как следует из схемы рис. 8.12 (штриховая линия), в этом случае в электронной стойке управления КИМ остаются только модули УСО СМ9104.12, которые через интерфейс ОШ подключаются к консольному процессору СМ2700.2805 из состава СМ1700. В этом случае в многозадачном режиме СМ1700 выполняет функции программного управления (подсистема ПУ) и обработки (подсистема ВК).

8.5. ПОЛЬЗОВАТЕЛЬСКИЕ ХАРАКТЕРИСТИКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВК СМ1700

Программное обеспечение ВК СМ1700 представлено следующими разделами:
операционные системы;
система программного диагностирования ВК СМ1700;
программное обеспечение для организации и управления базами данных;
программные средства телеобработки данных;
программные средства машинной графики и САПР.

Операционные системы представлены двумя системами: многофункциональная операционная система, поддерживающая виртуальную память, МОС ВП и диалоговая единая мобильная операционная система ДЕМОС-32.

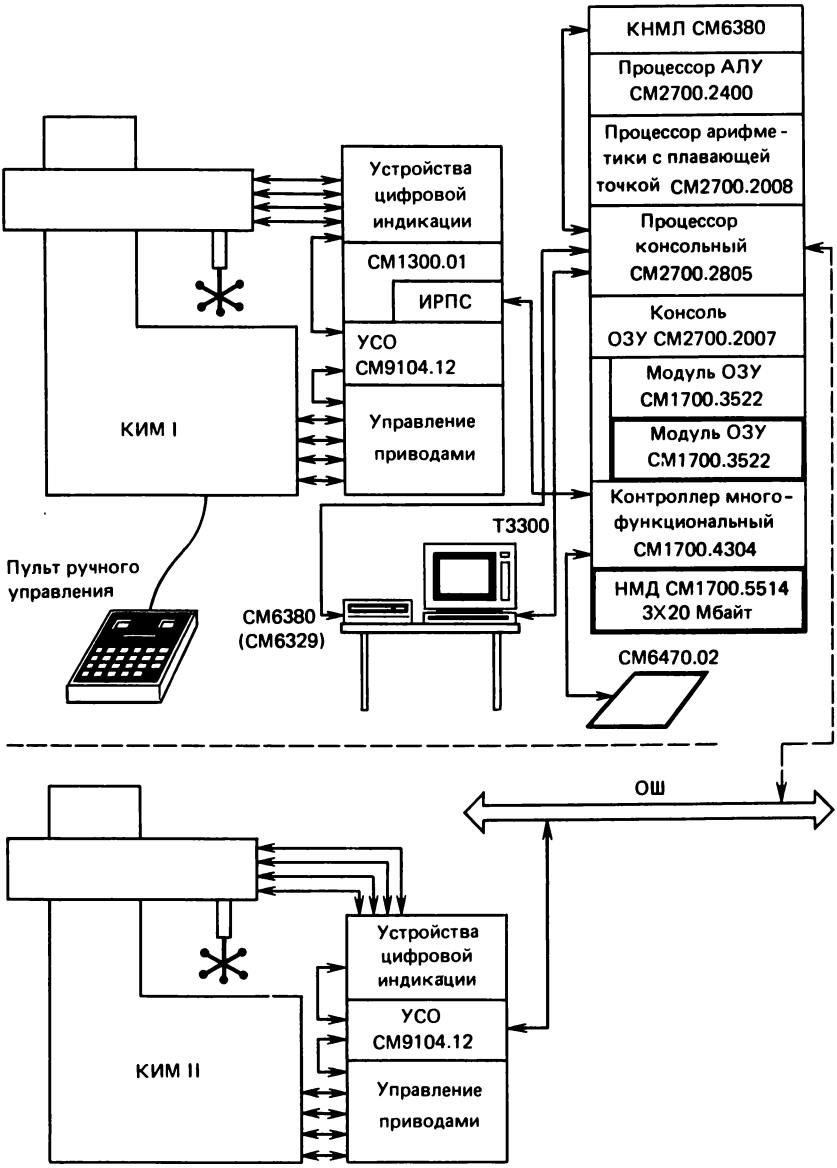


Рис. 8.12. СВКПУ КИМ на базе СМ1700

Система МОС ВП является базовой операционной системой, эффективно использует все архитектурные возможности СМ1700 и позволяет решать широкий класс информационных, вычислительных и управляющих задач. Такая система обеспечивает эффективное управление памятью, основанное на концепции

страничного распределения виртуальной и физической памяти между процессами. Процессы являются планируемой единицей работы в системе, для них распределяются вычислительные ресурсы. Планирование процессов осуществляется на основе 32 уровней приоритетов, которые разделяются между обычными процессами и процессами реального времени. Система обеспечивает выполнение процессов в интерактивном и пакетном режимах, а также в режиме реального времени.

Распределение ресурсов системы и комплекса базируется на значениях лимитов, квот и привилегий процессов, которые выполняются в различных режимах: ядра, управления, супервизора и пользователя. Ввод-вывод поддерживается на уровне физических устройств и на логическом уровне.

Система управления данными СУД-32 реализует верхний или логический уровень ввода-вывода, на котором выполняются операции, независимые от конкретного внешнего устройства. Запрос на непосредственное выполнение операции ввода-вывода на устройстве осуществляется посредством обращения к процедуре системного обслуживания QIO.

Система СУД-32 может быть использована при работе с любыми устройствами, включая терминалы и устройства печати, однако в первую очередь она предназначена для реализации программного интерфейса с устройствами с файловой структурой, такими как магнитные диски и ленты. При этом СУД-32 поддерживает последовательную, относительную и индексно-последовательную организацию файлов, различные форматы записей и способы доступа к записям в файлах.

Процедура QIO реализует нижний доступный пользователю физический уровень ввода-вывода. Она может быть вызвана явно из программы пользователя или неявно из системы управления данными. Обработка системой запроса QIO заканчивается вызовом драйвера устройства, т. е. программы, непосредственно выполняющей операцию ввода-вывода на устройстве.

Система МОС ВП позволяет использовать различные методы информационной связи между процессами: на основе разделяемой области данных и разделяемых файлов и с использованием механизма «почтового ящика» (виртуального устройства), с которым процессы могут обмениваться данными (читать и писать).

Связь между оператором и системой поддерживается с помощью диалогового командного языка. Кроме того, пользователю предоставляются средства для включения дополнительных команд оператора и создания собственных интерпретаторов командных языков.

Система МОС ВП поддерживает многопользовательскую защиту данных на уровне файлов, групп файлов и томов, предоставляет пользователю различные инструментальные средства: набор библиотек стандартных процедур и функций, средства подготовки текстовых файлов, отладочные средства, программы форматирования текстовых документов, программу сортировки записей файлов и др. Широкий набор системных обслуживающих программ позволяет эффективно выполнять вспомогательные функции в системе.

В целях сохранения преемственности ранее разработанного программного обеспечения МОС ВП обеспечивает информационную и программную совместимость с самой распространенной операционной системой 16-разрядной линии СМ ЭВМ — ОС РВ, что позволяет использовать разработанное в ОС РВ прикладное программное обеспечение на комплексе СМ1700.

В режиме совместимости с ОС РВ система МОС ВП предоставляет следующие возможности: выполнение непривилегированных задач, подготовленных в ОС РВ; поддержку томов с файловой структурой ОС РВ; выполнение непривилегированных команд программы связи с оператором ОС РВ.

Интерфейс пользователя с операционной системой в режиме совместимости осуществляется посредством эмуляции среды ОС РВ. Следует отметить, что эмуляция среды ОС РВ в МОС ВП является неполной. Например, не допускается выполнение привилегированных задач, а также задач, использующих директивы управления памятью или непосредственно обращающихся к внешней странице памяти.

Помимо программных средств эмуляции среды ОС РВ система МОС ВП включает программные компоненты ОС РВ, обеспечивающие создание пользовательских программ, работающих в режиме совместимости: текстовый редактор, программа работы с файлами, транслятор с языка макроассемблера, построитель задачи и др.

Система МОС ВП поддерживает следующие системы программирования: макроассемблер, Фортран, Кобол, Паскаль, СИ, Блисс-32, Бейсик, ПЛ/1, Модула, Корал, Диамс.

Операционная система ДЕМОС-32 разработана в рамках проекта по созданию Единой операционной среды (ЕОС) ДЕМОС. Основной задачей проекта ЕОС является обеспечение единого интерфейса пользователя и программиста для ЭВМ различных архитектур. При этом ставится условие обеспечения совместимости указанного интерфейса с интерфейсом широко распространенной за рубежом операционной системы UNIX. По своему назначению ДЕМОС является системой разделения времени с возможностью выполнения пакетных заданий. Средства обеспечения режима реального времени в основной состав системы не входят.

Система ДЕМОС имеет следующие особенности:

иерархическая древовидная схема управления процессами, в которой каждый процесс имеет свое адресное пространство, которое может быть использовано любым доступным образом (отсутствуют зарезервированные области для системных данных в виде таблицы, буферов);

отсутствие встроенного интерпретатора команд пользователя (практически любая программа может быть интерпретатором команд, причем для каждого пользователя своя);

иерархическая древовидная файловая система неограниченной глубины;

унифицированное представление файлов и периферийных устройств, что позволяет использовать одни и те же программы как для обычных файлов в файловой системе, так и для периферийных файлов (устройств);

отсутствие ограничений на представление данных в файлах (отсутствуют блоки записи, при этом любой файл рассматривается как пронумерованная последовательность байтов, что обеспечивает эффективную работу с файлами);

согласованное представление входных и выходных данных для программ позволяет создавать «поточные линии» по обработке данных без использования промежуточных временных файлов;

полный комплект программных средств для создания и обработки текстов, таблиц с большими и маленькими русскими и латинскими буквами;

наличие специальных средств описания терминалов позволяет ликвидировать зависимость программных средств (экранных редакторов и т. д.) от конкретных возможностей оборудования.

По своим функциональным характеристикам ДЕМОС-32 является расширением системы ДЕМОС для СМ4, что обусловлено в первую очередь архитектурой СМ1700 и набором ее внешних устройств. К отличительным особенностям системы ДЕМОС-32 следует отнести:

механизм страничного управления виртуальной памятью, что позволяет увеличить число активных процессов в системе и уменьшить объем подкачки процессов;

32-разрядная архитектура позволила снять ряд ограничений на размер ядра системы, что, в свою очередь, позволило улучшить функциональные характеристики, снизить затраты на работу самой системы, расширить функции ядра;

расширение адресного пространства позволило улучшить качественные и количественные характеристики программных компонентов системы (например, максимальный размер обрабатываемых файлов, программ и т. д.);

новая организация файловой системы (при сохранении ее логической структуры) на внешних носителях позволила на порядок поднять пропускную способность файловой системы, что является немаловажным фактором при обработке больших массивов данных, существенно поднять надежность хранения информации на внешних носителях.

Текущая версия системы ДЕМОС-32 предназначена в основном для создания инструментальных систем по разработке больших программных проектов, на ее основе могут быть созданы информационные, обучающие системы, системы подготовки высококачественной документации, системы распределенной обработки информации.

В последующих версиях системы ДЕМОС-32 планируется расширить поддержку внешних устройств, новых моделей 32-разрядных СМ ЭВМ. Будет достигнута совместимость с фактическим мировым стандартом UNIX SYSTEM-V. В состав системы будут включены реляционная СУБД, графические средства на базе графической корневой системы (ГКС), поддержка сетей, новые системы программирования (Пролог, Лисп).

Программное обеспечение для организации и управления базами данных включает многофункциональную информационную систему (МИС) СМ и комплексную автоматизированную реляционную систему (КАРС).

Система МИС СМ — программное обеспечение для централизованного управления базами данных, интерактивной и пакетной обработки данных на ЭВМ СМ1700 под управлением МОС ВП. Пользовательские данные хранятся либо в файлах МОС ВП, либо в виде базы данных сетевой структуры, управляемой системой Сеть-32. Доступ к данным возможен либо из программ на языках программирования (макро, Кобол, Бейсик, Фортран, Паскаль, ПЛ/1, СИ и др.), либо через интерактивную систему запросов.

Система МИС СМ состоит из четырех программных компонентов, которые могут использоваться как отдельно, так и совместно: система управления словарями Словарь-32, система управления базами данных Сеть-32; система управления формулярами СУФ-32; интерактивная система запросов Фобрин-32.

Система Словарь-32 предназначена для организации и ведения централизованного словаря данных. Все описания данных (метаданные) компонентов МИС

СМ, а также описания файлов на языке SLVL хранятся в словаре данных. В словаре хранятся следующие описания данных:

структуры файлов на языке SLVL (описание записи на SLVL может быть скопировано в программу на любом из языков программирования (Кобол, Бейсик, Фортран, Паскаль, ПЛ/1, СИ) в целях обработки одних и тех же файлов программами на нескольких языках);

записей, доменов, таблиц и процедур системы Фобрин-32;

схем, подсхем, схем управления доступом и схем хранения СУБД Сеть-32.

Словарь организован в виде иерархической структуры каталогов и объектов. Объекты находятся на самом нижнем уровне иерархии и содержат метаданные.

Авторизация метаданных осуществляется с помощью таблиц управления доступом. Обслуживающие программы Словарь-32 выполняют необходимые функции по поддержке целостности метаданных (защита и восстановление, проверка структуры, сбор статистической информации об изменении объектов).

Система СУБД Сеть-32 реализует предложения комитета КОДАСИЛ по базам данных. Язык описания данных (ЯОД) Сеть-32 позволяет эффективно реализовать сложные логические взаимосвязи данных пользователей, обеспечивает высокую реактивность прикладных программ, совместный доступ многих пользователей к базам данных, высокую степень защиты данных от разрушения и неавторизованного доступа.

Система Сеть-32 поддерживает одновременную работу 60 пользователей на одной ЭВМ с различными базами данных. Целостность данных обеспечивается средствами полного и частичного копирования баз данных, ведения журналов изменений и механизма захватов.

Прикладные программы могут обращаться к базам данных при программировании на любом языке, причем язык Кобол включает операторы языка манипулирования данными (ЯМД) Сеть-32, а для Фортрана имеется перекомпилятор ЯМД. Для остальных языков (макро, Паскаль, СИ, Бейсик, ПЛ/1) доступ к базам данных осуществляется через CALL-интерфейс. Для отладки прикладных программ, а также интерактивного взаимодействия с базами данных в состав Сеть-32 входит интерактивная подсистема запросов DBQ. Особенностью DBQ является возможность автоматического отображения структуры используемой подсхемы на видеотерминале в виде диаграммы Бахмана. Доступ к базам данных Сеть-32 возможен и через интерактивную систему запросов Фобрин-32.

Система СУФ-32 представляет собой набор программных средств для создания, хранения и использования в прикладных программах экранных формуляров (видеоформ), имеющих вид, максимально приближенный к виду документов (анкеты, бланки и т. п.). Видеоформа состоит из постоянной информации (заголовки, рамки и т. д.) и полей для ввода-отображения данных. Такая система позволяет задавать контроль данных, вводимых пользователем с терминала в поле, видеоатрибуты полей (реверс, яркость, мерцание), использовать символы двойной высоты и ширины, наложение видеоформ друг на друга.

Видеоформы СУФ-32 могут быть использованы в программах на всех языках программирования, а также при работе с данными через интерактивную систему запросов Фобрин-32.

Система Фобрин-32 — интерактивный процесс запросов, позволяющий формировать запросы к данным на специальном языке высокого уровня,

ориентированном как на программистов, так и на непрограммистов. Универсальность системы заключается прежде всего в том, что она позволяет обрабатывать данные, хранящиеся как в базах данных СУБД Сеть-32, так и в файлах последовательной, относительной и индексной организации, используя при этом унифицированный язык запросов.

Описания данных для Фобрин-32 могут быть написаны либо на собственном ЯОД системы, либо на ЯОД системы Словарь-32 SLVL. Эти описания, а также процедуры для обработки данных хранятся в централизованном словаре, обслуживаемом системой Словарь-32.

Система Фобрин-32 имеет в своем составе генератор отчетов, позволяющий формировать сложные форматированные отчеты. Для начинающих пользователей предусмотрены специальный режим сопровождения и средство генерации описаний, которые с помощью подсказки и меню позволяют быстро освоить ЯОД и ЯМД Фобрин-32.

При обработке данных (модификация, поиск) могут использоваться экранные формы, подготовленные с помощью системы СУФ-32. Использование CALL-интерфейса позволяет разрабатывать прикладные программы на языках Фортран, Кобол, Бейсик, Паскаль и других, расширяющих возможности Фобрин-32 по обработке данных.

Система МИС СМ функционирует на ЭВМ СМ1700 с объемом оперативной памяти не менее 2 Мбайт (если не используется СУБД Сеть-32, то 1 Мбайт). Программная часть системы занимает около 8 Мбайт на системном диске (на период генерации Сеть-32 требуется дополнительно 6 Мбайт на системном диске).

Система КАРС обеспечивает хранение и выборку алфавитно-цифровой информации для обработки в различных областях применения. Она поддерживает реляционную структуру данных: представление в виде отношений (таблиц), состоящих из строк и столбцов. В системе реализован язык SQL, который является наиболее мощным из языков, используемых в реляционных СУБД. Помимо ядра системы, интерпретирующего команды SQL, КАРС включает: системный интерактивный интерфейс (СИН), позволяющий выполнять команды SQL в интерактивном режиме и управлять форматированием выходной информации; систему интерактивных приложений (СИП), обеспечивающую обработку данных через экранные формы, ориентированные на конечного пользователя; генератор отчетов, выполняющий форматирование и вывод информации из базы данных с включением вспомогательного текста; обслуживающие программы, обеспечивающие загрузку, реорганизацию и восстановление базы данных; интерфейсы с языками программирования высокого уровня.

Система КАРС обеспечивает мультидоступ к данным с защитой данных от одновременного обновления, от неавторизованного доступа. Все изменения данных заносятся в файл изменений, что обеспечивает возможность восстановления базы данных после аппаратных сбоев. Физическая структура базы данных обеспечивает эффективное хранение данных и вместе с тем возможность быстрого поиска. Для этого используются индексирование и кластеризация таблиц.

База данных КАРС может размещаться в нескольких файлах, что позволяет более гибко использовать физическое пространство внешних запоминающих устройств. Простота взаимодействия с системой КАРС, наличие компонентов, ориентированных на конечного пользователя, позволяют существенно сократить сроки освоения системы и ускорить создание на ее основе информационных систем.

8.6. ВОПРОСЫ РЕАЛИЗАЦИИ ТИПОВЫХ МНОГОМАШИННЫХ И СЕТЕВЫХ КОНФИГУРАЦИЙ

В состав программных средств телеобработки СМ1700 в настоящее время входят системы:

программного обеспечения распределенных сетей — ТРАЛ;

программного обеспечения локальных сетей — МАГИСТР;

программного обеспечения для организации распределенных многомашинных комплексов на базе СМ1700 и ЕС ЭВМ — ЭМУЛЯТОР;

программного обеспечения для сетей СМ ЭВМ с малыми ресурсами — СЕТЬ МИНИ;

однородная операционная среда ОС ДЕМОС для сети ЭВМ различных типов — ДЕМОН;

программного обеспечения локальных сетей кольцевого типа — КОЛОС.

Система программного обеспечения (СПО) ТРАЛ является базовым сетевым программным обеспечением для создания многомашинных систем автоматизации и сетей ЭВМ на базе СМ1700. Она позволяет создавать однородные распределенные сети ЭВМ СМ1700 любой топологии и размерностью до 1024 узлов. Использование пакета программ (ПП) для создания однородных сетей СМ ЭВМ (СПО СЕТЬ СМ) обеспечивает включение ЭВМ типа СМ1420 (СМ4, СМ1300, СМ1600 и программно совместимых с ними) в качестве узлов в создаваемую распределенную сеть. Работает СПО ТРАЛ под управлением МОС ВП.

В создаваемой на основе СПО ТРАЛ вычислительной сети реализуются следующие основные функции:

взаимодействие пользовательских программ, выполняемых в различных узлах сети;

транспортировка сообщений от программы-источника к программе-адресату в сети с количеством узлов до 1023;

доступ из программ и с терминалов к файлам в удаленных узлах;

обмен файлами между различными узлами сети;

запуск удаленных программ в режиме диалога и дистанционная пакетная обработка;

обмен сообщениями между терминалами различных узлов сети;

адаптивное управление потоками данных и функционированием сети в целом;

использование терминалов, подсоединенных к различным узлам, как сетевых, т. е. допускающих логическую связь с другими узлами;

отображение информации о функционировании любого элемента сети;

тестирование сетевого оборудования и программного обеспечения.

Связь между узлами обеспечивается асинхронным и синхронным телекоммуникационным оборудованием из номенклатуры СМ ЭВМ.

Благодаря используемому протоколу управления информационным каналом достигается безошибочность передачи информации между узлами сети, оптимизация процесса передачи, проверка каналов связи, сбор статистики и т. д.

Система программного обеспечения МАГИСТР является базовым сетевым программным обеспечением для создания на базе ЭВМ СМ1700 локальных сетей магистрального типа с множественным доступом, с обнаружением несущей и наложений.

Телекоммуникационное оборудование, используемое для создания локальной сети (контроллеры локальной сети магистрального типа, коаксиальные кабели с устройствами преобразования сигналов), позволяет осуществлять передачу со скоростью до 10 Мбит/с.

Информационно и функционально СПО МАГИСТР совместима с СПО ТРАЛ и допускает их объединение. Существенным новым фактором, расширяющим возможности системы, является использование быстродействующей магистрали для организации информационной связи и создания локальной сети. Высокая скорость передачи обеспечивает качественно новый уровень взаимодействия ЭВМ.

Система программного обеспечения ЭМУЛЯТОР предназначена для организации распределенных многомашинных комплексов на базе ЕС ЭВМ и СМ1700. При этом МОС ВП расширяется программными средствами, обеспечивающими обмен информацией между СМ1700 и ЕС ЭВМ.

Путем эмуляции терминальной системы ЕС 7920 в СПО ЭМУЛЯТОР реализуются следующие основные функции:

функционирование СМ1700 в качестве удаленных абонентских пунктов ЕС ЭВМ;

обмен данными между программами, функционирующими в СМ и ЕС ЭВМ.

Система программного обеспечения Сеть мини предназначена для организации сетевого взаимодействия разнотипных СМ ЭВМ, функционирующих под управлением различных операционных систем. Комплекс программ, составляющих СПО СЕТЬ МИНИ, функционирует на СМ1700 под управлением МОС ВП, на СМ1420 (СМ4, СМ1300) под управлением ОС РВ и РАФОС, на СМ1800—ОС 1800, на СМ 1810 и ЕС 1840—МДОС.

Система программного обеспечения СЕТЬ МИНИ для организации взаимодействия ЭВМ использует терминальные линии связи и обеспечивает:

эмуляцию терминала удаленной ЭВМ, т. е. установление логической связи терминалов одной ЭВМ с другими узлами сети;

передачу символьных и двоичных файлов между узлами сети по командам оператора.

Система программного обеспечения СЕТЬ МИНИ обладает следующими достоинствами:

надежностью (достоверностью) передачи информации;

возможностью объединения в сеть различных типов ЭВМ с различными операционными системами и с соответствующим преобразованием формата символьной информации;

простотой аппаратной стыковки различных ЭВМ, вследствие чего СПО является дешевым и технически простым способом организации сетевого взаимодействия;

передачей последовательности повторяющихся символов в сокращенной (уплотненной) форме.

Однородная операционная среда ДЕМОН предназначена для использования в качестве базовой сетевой среды при создании сетей разнотипных СМ ЭВМ (СМ1700 и СМ1420), работающих под управлением ОС ДЕМОС. В этой операционной среде обеспечивается реализация следующих функций:

пересылки файлов между узлами сети;

терминального доступа к удаленным узлам, т. е. организации логической связи терминала с удаленной сетевой СМ ЭВМ.

Система программного обеспечения КОЛОС предназначена для использования в качестве базового сетевого программного обеспечения при создании локальных сетей кольцевого типа из разнотипных СМ ЭВМ и оконечного оборудования (терминалы, печатающие устройства). Оконечное оборудование и ЭВМ включаются через станции локальной сети кольцевого типа СЛК СМ в кольцевую локальную сеть. Объединяемые в сеть, СМ ЭВМ могут иметь различное прикладное и функциональное назначение, а также работать под управлением разных операционных систем. При этом обеспечивается реализация следующих основных функций:

межмашинного взаимодействия до 125 ЭВМ со скоростью передачи по кольцу 500 000 бит/с, а между СЛК СМ и присоединенным оборудованием до 19 200 бит/с;

пересылки файлов между узлами локальной сети по командам оператора;

терминального доступа к удаленным узлам, т. е. организации логической связи терминала одной СМ ЭВМ с другой сетевой СМ ЭВМ;

сетевого терминального доступа, т. е. организации логической связи сетевой СМ ЭВМ с терминалом, подключенным непосредственно к СЛК СМ.

Базовое программное обеспечение автоматизированных рабочих мест на базе СМ1700 (БПО АРМ СМ1700) предназначено для использования на графических комплексах, а также широко применяется в САПР для работы пользователей в режимах

реального времени, разделения времени и в пакетном режиме. Базовое программное обеспечение АРМ СМ1700 включает базовые средства машинной графики, средства поддержки проблемно-ориентированных графических пакетов, один из графических проблемно-ориентированных пакетов и драйверы графических устройств, входящих в состав комплекса АРМ СМ1700.

Ориентировано БПО АРМ СМ1700 на работу в МОС ВП и включает следующие функциональные компоненты:

драйверы графических устройств;

программные средства машинной графики на основе графической корневой системы (ГКС);

пакеты программ машинной графики.

Базовое программное обеспечение АРМ СМ1700 обеспечивает возможность включения широкого класса проблемно-ориентированных пакетов прикладных программ, содержащих пакеты и программные средства, предназначенные для проектирования изделий радиоэлектроники, машиностроения, БИС и др.

Базовые программные средства машинной графики (БПС МГ) на основе ГКС предназначены для поддержки ввода-вывода графической информации посредством графических устройств, функционирующих в составе комплекса СМ1700.

Область применения БПС МГ обусловлена необходимостью представления входной и выходной информации, используемой в проблемно-ориентированных задачах, в графической форме.

Использоваться БПС МГ могут как вспомогательные (инструментальные) средства для разработки прикладных программ в системах:

автоматического проектирования в электронике, машиностроении, строительстве и др.;

автоматизации научных исследований;

автоматизированных управлений технологическими и производственными процессами;

отображения информации в медицине, метрологии, геофизике, картографии и т. д.

Базовые программные средства МГ допускают возможность подключения силами пользователя программных модулей, обеспечивающих функционирование различных графических устройств.

Отличительной особенностью БПО АРМ СМ1700 является проблемная ориентация путем включения в состав БПО одного из проблемно-ориентированных пакетов программ, предназначенных для сквозного автоматизированного проектирования.

В настоящее время ведется разработка следующих проблемно-ориентированных пакетов программ:

автоматизированного проектирования БИС, который предназначен для использования в проблемно-ориентированных комплексах АРМ СМ1700, ориентированных на автоматизированное проектирование и моделирование БИС, а также средства диалога

и получения твердых копий результатов проектирования заказных и полузаказных БИС;

автоматизированного проектирования изделий машиностроения, включающих диалоговую графическую систему трехмерной графики, системы вычерчивания чертежей, разработки и выдачи управляющих программ для станков с ЧПУ;

автоматизированного проектирования сложных поверхностей, включающих программы проектирования изделий для авиа-, авто- и судостроения с выдачей результатов проектирования на станки с ЧПУ;

автоматизированного архитектурно-строительного проектирования, включающего программы проектирования (в двух- и трехмерном изображении) архитектурных и строительных конструкций, зданий, сооружений и др.;

прочностных расчетов методами конечных элементов, включающих программы прочностных расчетов в интерактивном режиме с графическим отображением конструкций;

автоматизированного проектирования печатных плат, включающих программы автоматизированного проектирования (размещение компонентов, трассировку соединений и контроль) многослойных печатных плат для широкого класса пользователей.

8.7. ОРГАНИЗАЦИЯ ДИАГНОСТИРОВАНИЯ ВК СМ1700

Развитие современных средств вычислительной техники связано с повышением сложности и трудоемкости процесса обнаружения и поиска неисправностей в ВК. В связи с этим создание высокоэффективных средств диагностирования являлось одной из важных задач, связанных с разработкой ВК СМ1700. Характерные для малых ЭВМ аппаратные и стоимостные ограничения выдвинули на передний план развитие программных средств диагностирования, разработка которых проводилась одновременно с разработкой технических средств комплекса. Это позволило при относительно небольшом объеме дополнительной аппаратуры обеспечить высокую программную диагностируемость. Под программной диагностируемостью понимается наличие в СМ1700 аппаратных средств, обеспечивающих поддержку программных средств диагностирования при:

создании проверяемых форм хранения диагностической информации о состоянии комплекса;

выполнении отдельных проверок, критичных по времени или затруднительных с программной точки зрения;

повышении разрешающей способности.

Аппаратные средства диагностирования ВК СМ1700 включают:

контроль непротиворечивости и отсутствия ошибок, который определяет неправильное использование инструкций, некоторые перемежающиеся и постоянные ошибки в работе аппаратуры

и запрещенные арифметические условия (переполнение, антипереполнение, деление на нуль);

контроль выполнения инструкций вызова подпрограмм и возврата из них на правильность сохранения и восстановления общих регистров;

реализацию в системе команд специальной инструкции CRC, обеспечивающей быстрое поблочное вычисление кода контрольной суммы в задачах телеобработки;

автоматическое исправление одиночных ошибок и обнаружение двойных ошибок в оперативной памяти с помощью корректирующего кода;

реализацию инструкций для контроля обращения процесса к оперативной памяти;

контроль по паритету, который охватывает передачу данных, буфер трансляции виртуальных адресов, микропрограммы, а также внутренние шины центрального процессора;

введение таймера с высоким разрешением, который обеспечивает проверку функций, зависящих от времени;

введение специальных регистров, позволяющих отображать ошибочные состояния в комплексе.

Кроме того, в СМ1700 включены аппаратные средства, предусматривающие выборочное исключение аппаратуры (диспетчер памяти, буфер трансляции, процессор с плавающей точкой) из работы для облегчения локализации неисправностей.

Система программного диагностирования СМ1700 с учетом структуры центральной части ВК СМ1700 реализована по принципу расширяющихся областей (сред) диагностирования. Область диагностирования характеризуется совокупностью аппаратных и программных средств, предоставляющих диагностическим программам определенные функциональные возможности по организации процесса диагностирования и выбору тестовых воздействий. Принцип расширяющихся областей в настоящее время является наиболее эффективным при реализации процедуры самодиагностирования для ВК. Согласно этому принципу в каждой области диагностирования ВК СМ1700 более низкого уровня проверяется аппаратное ядро, т. е. необходимый минимум работоспособной аппаратуры области более высокого уровня. Это позволяет не только упростить процесс поиска неисправностей, но и в значительной степени избежать влияния некорректных неисправностей на этот процесс.

Структура программного диагностического обеспечения ВК СМ1700 представлена на рис. 8.13.

Программные средства диагностирования имеют иерархическую структуру, в которой можно выделить шесть уровней: 1, 2R, 2, 3, 4 и 5. Диагностические программы каждого более высокого уровня требуют для своего запуска и выполнения большего аппаратного ядра и расширяют область аппаратуры, используемой для организации процесса диагностирования.

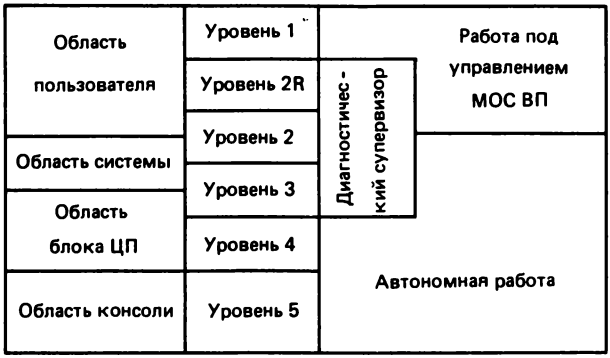


Рис. 8.13. Структура программной диагностики ВК СМ1700

Первый уровень программной диагностики представлен пакетом программ, выполняемых под управлением операционной системы, который обеспечивает комплексную проверку работоспособности внешних устройств и основных программных компонентов системы.

Следующие четыре уровня образуют систему диагностических программ для проверки работоспособности и поиска неисправностей технических средств. На этих уровнях решается задача обеспечения максимальной полноты проверки и детализации задачи поиска неисправностей. При этом детализация достигается взаимной увязкой различных диагностических программ по уровням и внутри одного уровня, а также разбиением каждой диагностической программы на тесты, подтесты и выделением в них элементарных проверок.

Диагностические программы уровней 2, 2R и 3 выполняются под управлением диагностического супервизора (ДС). Различие диагностических программ этих уровней связано с использованием ими ресурсов МОС ВП.

Диагностический супервизор реализует командный язык, который позволяет выполнить широкий набор функций по настройке системы на заданное соединение проверяемых устройств; загрузке диагностических программ с различных носителей; запуску и управлению выполнением диагностических программ; отладке и модификации загруженных в память программ; выдаче справочной информации о системе программного диагностирования и о составе поддерживаемых ее технических средств.

Создание ДС с возможностями, близкими к возможностям операционных систем, имеет важное значение для повышения эффективности системы программного диагностирования в целом. За счет предоставления широкого набора сервисных функций со стороны ДС уменьшается среднее время поиска неисправности и время профилактических осмотров. За счет вынесения общих для всех диагностических программ процедур в ДС значительно

экономятся затраты на их разработку и унифицируется управление отдельными тестовыми проверками.

Унификация управления наряду с гибкостью позволяет организовать различные специальные режимы проведения диагностирования (например, посредством многократных зацикливаний отдельных групп элементарных проверок (тестов, подтестов) для повышения полноты диагностирования в классе перемежающихся неисправностей, а также для применения диагностических программ на заключительных этапах наладки машины), а разрешенный в ДС режим покомандного выполнения диагностических программ — детализировать момент возникновения ошибки.

Диагностические программы уровня 2R выполняются только с использованием ресурсов операционной системы (прежде всего системных драйверов) и предназначены для проверки работы внешних устройств в мультипрограммном режиме, а также правильности передачи данных с учетом различных протоколов, которые поддерживаются в системе. Важной характеристикой уровня 2R является возможность выполнения процедуры диагностирования с помощью программ данного уровня параллельно с другими процессами в системе. Это позволяет не останавливать работу ЭВМ на время профилактических осмотров и поиска отдельных неисправностей, не нарушающих работоспособности вычислительной системы в целом. Выполнение проверки уровня 2R на завершающих этапах диагностирования СМ1700 обеспечивает работоспособность технических средств среды пользователя.

Диагностические программы уровня 2 выполняются как с использованием ресурсов операционной системы, так и без них под управлением автономно работающего ДС. Первый режим работы предоставляет возможность так же, как и на уровне 2R, выполнять процедуру диагностирования как пользовательский процесс, не останавливая работу машины. Второй режим предусмотрен на случай, если возникшие неисправности не позволяют запустить операционную систему. В первом режиме при выполнении диагностических программ используются системные драйверы, а во втором — драйверы ДС (диагностические драйверы).

Диагностические программы уровня 2 применяются для тренировки ЦП и функциональной проверки внешних устройств. Они завершают диагностирование среды системы и совместно с программами уровня 2R подготавливают работу среды пользователя.

Диагностические программы уровня 3 выполняются под управлением автономного ДС. Эти программы осуществляют комплексную проверку ЦП, проверку привилегированных инструкций, различных режимов ЦП, а также проверку основных функций внешних устройств. На уровне 3 возможен доступ к периферийным устройствам на уровне регистров, что обеспечивает максимальную разрешающую способность при диагностировании, необходимую для ремонта периферийных устройств.

Диагностические программы уровня 4 выполняются автономно без ДС. На этом уровне обеспечиваются только самые простые средства загрузки, управления и формирования сообщений об ошибках. Поэтому несмотря на то, что режим работы уровня 4 предоставляет широкие возможности для обнаружения неисправностей, диагностические программы этого уровня используются ограниченно, только для получения такой информации, которую нельзя получить при диагностировании на других уровнях. В настоящее время программы уровня 4 ограничены проверкой набора инструкций, необходимого для работы ДС.

Диагностические средства уровня 5 представлены системой диагностических микропрограмм, выполняемых под управлением диагностического микромонитора, и ПЗУ-резидентным микротестом, запускаемым при включении питания.

На этом уровне решаются следующие две важнейшие задачи диагностирования СМ1700: с одной стороны, минимизируется аппаратное ядро, необходимое для организации процесса самодиагностирования, с другой — достигается максимальная разрешающая способность.

Диагностические микропрограммы, управляемые микромонитором, делятся на две группы. Программы первой группы запускаются из ЗУП консольно-диагностического процессора (КДП) и выполняют проверку аппаратуры ЦП, необходимой для работы второй группы программ. Диагностические микропрограммы второй группы запускаются из управляющей памяти АЛП и проверяют другие составляющие ЦП. Такая разбивка позволяет уменьшить минимально необходимое аппаратное ядро для начала процесса самодиагностирования.

Значение микромонитора для диагностических микропрограмм аналогично значению ДС для диагностических программ уровней 3, 2, 2R. Микромонитор предоставляет широкий набор сервисных функций по загрузке, запуску и управлению выполнением микропрограмм, по модификации содержимого микропрограммно доступных схем ЦП, позволяет организовать выполнение группы тестов с возможностью зацикливания, обеспечивает выполнение микропрограмм по микрокомандам.

Самый начальный этап самодиагностирования ВК СМ1700 определяется ПЗУ-резидентным микротестом, который выполняется КДП, поскольку не требует средств загрузки микропрограмм. В блоке ЦП имеется переключатель, разрешающий отключение микротеста.

Реализация резидентных тестов самодиагностирования представляет собой эффективное средство проверки внутренних компонентов комплекса при минимальном влиянии со стороны других составляющих комплекса. Организованный таким образом доступ к внутренним точкам объекта диагностирования позволяет увеличить полноту и разрешающую способность средств программного диагностирования.

Для удобства выполнения процедуры диагностирования ВК СМ1700 может быть использована программа «Экспресс-проверки». Это специальная управляющая программа, позволяющая пользователям выполнять диагностирование комплекса с использованием диагностических и микродиагностических программ. Программа «Экспресс-проверки» может выполняться в автоматическом режиме без вмешательства пользователей или с использованием упрощенного диалога, где можно в режиме «меню» выбрать один из возможных способов диагностирования комплекса.

Стратегия диагностирования ВК СМ1700 определяется структурой системы программной диагностики и эффективностью отдельных уровней системы.

Можно выделить две основные задачи, встречающиеся в процессе эксплуатации вычислительного комплекса. Первая задача, с которой часто сталкиваются во время профилактических осмотров, называется проверкой работоспособности. При решении этой задачи не делается никаких предположений о наличии и характере неисправности. Хотя проверка работоспособности начинается уже с выполнения ПЗУ-резидентного микротеста по включению питания, действительным ее началом следует считать выполнение «экспресс-проверки», которая запускается по упрощенному диалогу с консоли и в дальнейшем не требует вмешательства оператора. Если на этом этапе неисправность не обнаружена, то дальнейшая проверка осуществляется с помощью средств программного диагностирования уровней 5 и 3, не вошедших в «экспресс-проверку», а затем последовательно уровней 2, 2R и 1. Если на начальном этапе неисправность обнаружена, то необходимо перейти к задаче поиска неисправностей. В зависимости от места неисправности (ЦП или периферийное устройство) используются микродиагностические программы уровня 5 или последовательно диагностические программы уровней 3 и 2.

Задача поиска неисправности является второй задачей диагностики, которая обычно возникает при обнаружении ошибок функционирования ВК под управлением операционной системы. На первом этапе следует использовать, если это возможно, программные средства МОС ВП (уровень 1), которые выполняются как обычные пользовательские процессы. Диагностические программы уровней 2R и частично 2 позволяют определить неисправную подсистему. Дальнейшая локализация неисправности осуществляется под управлением автономного ДС с помощью диагностических программ уровней 3 и частично 2, а затем микродиагностических программ.

Если операционная система не загружается, то делается попытка загрузить ДС с основного (магнитный диск) устройства загрузки или, в случае неудачи, вторичного (консольное устройство). Если это также не удается, то следует выполнить микродиагностические программы и программы уровня 4.

Кроме рассмотренных средств программной диагностики определенное место в процедуре диагностирования ВК СМ1700 занимают программные средства проверки функционирования комплекса под управлением МОС ВП. Эти программные средства предоставляют возможность регистрации аппаратных и программных сбоев в системе, обеспечивают сбор статистики о фун-

кционировании системы, позволяют проводить анализ аварийного и текущего состояний системы.

Особое место среди программных средств занимает пакет программ комплексной проверки функционирования вычислительной системы СМ1700 «Эксперт», который используется как первый уровень системы программного диагностирования. Проверка системы с помощью пакета «Эксперт» не дает исчерпывающей информации об исправности системы в целом, однако успешное выполнение проверок гарантирует готовность системы к эксплуатации.

Программы, входящие в пакет, создают имитацию нагрузки в системе, обеспечивая комплексную проверку работоспособности внешних устройств и основных компонентов системы. Пакет может использоваться для предварительного диагностирования комплекса, а также для демонстрации системных возможностей.

Развитие системы программного диагностирования предусматривает построение специальных центров обслуживания, оснащенных экспертными системами для автоматизации поиска неисправностей и связанных линиями связи с удаленными ВК СМ1700.

8.8. ПОКАЗАТЕЛИ ПРЕДЕЛЬНОЙ ПРОИЗВОДИТЕЛЬНОСТИ СМ ЭВМ

История развития ЭВМ — это история борьбы разработчиков и пользователей вычислительной техники за увеличение технико-экономических показателей и прежде всего производительности. С развитием ЭВМ и расширением областей применения СВТ качественно изменялись задачи и режимы обработки информации с применением ЭВМ, принципиально изменялись архитектура и способы организации вычислительного процесса.

Производительность ЭВМ является комплексным синтетическим показателем технической эффективности применения СВТ. Достижимый уровень производительности должен коррелироваться с затратами времени $T_{об}$ заданного множества заявок $\{B_i\}$ в определенных заданных условиях эксплуатации. Этот показатель определяется архитектурой ЦП (набором команд и временем выполнения отдельных команд, иерархией внутренней и внешней памяти, степенью распараллеливания (совместимостью) отдельных этапов обработки информации, пропускной способностью интерфейсов, системой прерываний и т. д.), набором периферийных устройств в конкретном комплексе, совершенством операционных систем и прикладного программного обеспечения, надежностью вычислительного комплекса и т. п.

Исходными данными для такого анализа могут служить три вида (уровня) элементарных показателей производительности:

время выполнения отдельных операций (базовых и арифметических с плавающей точкой — с одинарной и двойной точностью);

производительность на смеси алгоритмических действий — MIX для реального времени, MIX научно-технических и MIX экономических расчетов;

производительность на типовых задачах (например, для научно-технических расчетов типа ГАММ, Whetstone и т. д.).

На первых этапах развития массовых ЭВМ (60-е годы) в силу совпадения архитектуры большинства ЭВМ достаточно было определить тактовую частоту работы процессора или время выполнения одной из команд. Обычно это было время выполнения сложения двух чисел.

В 70-е годы широкое применение нашли оценки производительности, получившие название MIX-характеристики. Они определяют число осредненных операций (простейших алгоритмических действий) $P_д$ или $P_к$, соответствующее им число осредненных машинных команд, выполняемых в секунду. Осреднение команд, проведенное с учетом достаточно большого числа задач, позволяет определить следующую зависимость для расчета MIX-характеристики:

$$P_к = \frac{\sum_{i,j} a_i b_{ij}}{\sum_{ij} a_i b_{ij} t_{ij}},$$

где a_i — вес алгоритмического действия i-го типа, с; b_{ij} — повторяемость команды j-го типа в i-м действии; t_{ij} — время выполнения команды j-го типа в i-м действии, с.

На практике наибольшее распространение получили смеси алгоритмических действий, предложенные Гибсоном. Раздельно определяется производительность для обработки информации в реальном времени для научно-технических и экономических расчетов. Время выполнения ряда команд и MIX-характеристики для СМ1700, основных моделей СМ ЭВМ, а также ряда моделей ЕС ЭВМ и некоторых моделей зарубежных фирм (по опубликованным данным) приведено в табл. 8.3. (В этой же таблице приведены и другие показатели производительности, анализируемые ниже.)

Однако качественное совершенствование архитектуры ЭВМ и прежде всего расширение системы команд, обеспечивающих аппаратное выполнение сложных логических и арифметических действий с повышенной точностью, совершенствование системы адресации, использование частичного или полного совмещения операций, осуществление более эффективного управления процессами обработки информации приводят к тому, что ни время выполнения отдельных команд, ни MIX-характеристики не могут

Таблица 8.3. **Характеристики производительности**

Показатель производительности	Единицы измерения	СМ1420	СМ1700	VAX11/750	VAX11/780	ЕС1035	ЕС1045	ЕС1055	ЕС1060
$P_к$	млн коротких команд/с	1,0	1,0	–	–	1,39	1,70	1,06	3,22
P_{W1}	тыс. опер. Whetstone/с (32 разр.)	90	323	701	1165	–	500	285	736
P_{W2}	тыс. опер. Whetstone/с (64 разр.)	72,9	212	513	750	–	–	–	–
P_{W4}	тыс. опер. Whetstone/с (128 разр.)	–	179	136	194	–	–	–	–
P_{W6}	тыс. опер. Whetstone/с (256 разр.)	--	67	39	46	–	–	–	–
$P_{W\,ср}$	тыс. опер. Whetstone/с	76,3	234	550	823	–	–	–	–
$П^{W}_{отн}$	–	1	3,05	7,2	10,8	–	5,55	3,16	8,2
$П^{W}_{экв}$	млн коротких команд СМ1420/с	1,0	3,05	–	–	–	–	–	–
$T_г$	с	197	42	12,8	7,6	–	30,8	–	20,5
$P_{Ги}$	тыс. опер. Гибсона/с	175 (212*)	245	–	–	269	513	282	628

* С процессором СМ2420, включающим кэш-память.

дать адекватную оценку производительности современных ЭВМ. Производительность ЭВМ, определяемая как число типовых заявок на обработку информации, выполняемых в единицу времени в заданных условиях эксплуатации, может быть оценена только моделированием процесса обработки информации на ЭВМ в целом.

Для того чтобы уточнить характер и набор типовых заявок для моделирования процесса обработки информации, рассмотрим корреляцию ряда основных архитектурных особенностей 32-разрядной ЭВМ СМ1700 и оценок ее производительности.

Из отмеченных в 8.1 основных качественных направлений совершенствования архитектуры СМ1700 на производительность прежде всего влияют:

1. Расширение системы команд за счет введения специальных команд типа:

POLY, позволяющей аппаратно вычислять полином 4-й степени, что значительно сокращает время исполнения стандартных программ типа SINE и COSINE;

CEISY, реализующей многопараметрическое управление циклами DO в программах на Фортране;

LOAD/STORE CONTEXT, реализующей быстрое переключение в операционной системе с одного процесса на другой при диспетчеризации мультипроцессорной обработки.

2. Развитие системы адресации за счет:

трехадресных команд, которые обеспечивают сохранение исходных операндов, значительно снижают затраты на пересылку информации и увеличивают долю использования максимально эффективных регистровых команд;

четырех-, пяти- и шестиоперандовых команд, позволяющих оптимизировать и упрощать программирование арифметических и логических преобразований;

использования в относительном и косвенно-относительном режимах адресации трех модификаций смещения (байт, слово, двойное слово) и введения двух новых (по отношению к СМ1420) режимов (с коротким литералом – не более 6 разрядов и индексного), которые позволяют сокращать требуемые объемы памяти и оптимизировать обработку массивов;

расширения типов обрабатываемых данных (байт, 16-, 32-, 64-, 128-разрядные слова с фиксированной и плавающей точкой, битовые поля переменной длины, символьные и десятичные строки), которые значительно сокращают затраты времени на преобразования информации, оптимизируют использование ресурсов под заданную точность и тип данных.

3. Частичное совмещение операций выборки и выполнения команд в результате введения 4-байтового буфера предвыборки в арифметико-логическом процессоре. Только благодаря этому среднее время выполнения коротких команд уменьшается на 30—50%. Зацикливание одной команды при определении времени ее выполнения (рекомендуемое в методиках для ЭВМ с «обычной» архитектурой) неприемлемо, так как не обеспечивает работу механизма предвыборки. При определении времени выполнения отдельных команд СМ1700 в условиях, соответствующих реальному режиму обработки информации, необходимо зацикливать последовательности не менее двух команд.

На рис. 8.14 приведены результаты замеров времени выполнения команд в следующих пяти режимах зацикливания последовательностей команд:

I — измеряемая команда;

II — длинная-измеряемая-длинная команды;

III — длинная-измеряемая-короткая команды;

IV — короткая-измеряемая, оператор цикла;

V — короткая-измеряемая-короткая.

В табл. 8.4 даны времена выполнения ряда характерных команд (мкс) СМ1700, в том числе команд плавающей арифметики. При этом варьировался способ адресации трехадресных операндов, включая косвенную адресацию трех адресов.

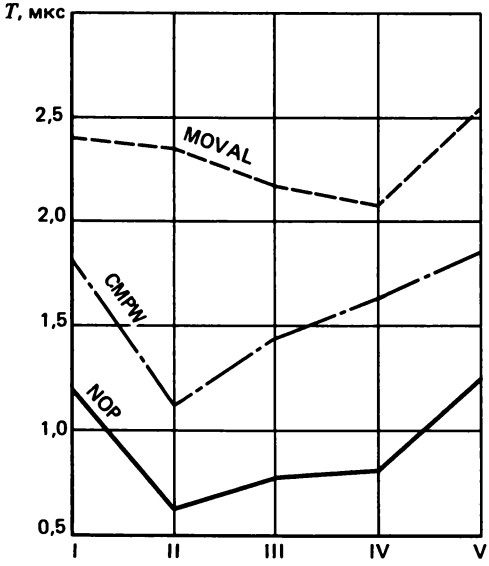


Рис. 8.14. Экспериментальные результаты времени выполнения команд

Таблица 8.4. **Время выполнения команд СМ1700 для различных способов адресации**

Команда	Время выполнения команд, мкс			
	R	1(R)	2(R)	3(R)
MUVL	1,69	1,85	1,85	—
BEQLU*	0,84	—	—	—
JMP	0,84...4,5**	—	—	—
ADOW3	1,69...1,75	2,55...2,64	3,43...3,52	5,6...5,8
SUBW3				
MULW3	10,7...10,9	11,36...11,58	12,01...12,16	14,15...14,5
DIVW3	38,7...39,4	39,2...39,8	39,9...40,5	41,6...42,3
NOP*	0,6...1,2	—	—	—
CMPW	1,1...1,8	1,5...2,3	2,8...3,5	—
CMPF	3,98...4,6	5,93...3,6	7,05...7,8	—
CMPB	1,09...1,8	2,15...2,8	2,86...3,5	—
MUVC3***	450	1520	—	—
ASHL	8,2...8,9	9,9...10,6	10,54...11,2	—
BISW3	2,36...3,0	3,37...4,01	4,56...5,3	—
MUVAL	3,8	4,16...4,25	—	—
ADDF2	4,51...5,13	—	—	8,81...14,25****
ADDF3	4,74...5,483	5,8...5,48	7,03...7,4	9,5...9,73
DIVF3	10,4...10,5	11,4...11,7	12,6...12,8	15,1...15,5
MULF3	9,2...9,36	10,5...10,6	11,6...11,8	14,07...14,31

* Нерегистровая операция.
** Передача управления на другую страницу.
*** Пересылка поля 500 слов из одной области памяти в другую.
**** Работа с памятью.

Учитывая вышесказанное, для адекватной оценки производительности вычислительных комплексов целесообразно выбрать ряд типовых программ, так называемых *бенчмарков*. Такая типовая программа должна удовлетворять следующим требованиям:

представлять смесь элементарных заданий, соответствующих среднестатистическому набору на представительном наборе программ для анализируемой области применения ЭВМ;

должна быть синтетической, т. е. включать несвязанные отдельные программные модули, чтобы исключить возможность повторного использования промежуточных результатов при оптимизации в режиме трансляции, сокращающих реально исполняемое число инструкций;

должна быть написана на распространенном языке высокого уровня, обеспечивающем ее прогон на широком классе ЭВМ для построения сравнительных оценок их производительности.

Для оценки производительности на научно-технических заданиях средней сложности широко в мире используется программа Whetstone. Удовлетворяющая всем приведенным выше требованиям, эта программа включает 10 модулей, выполняющих среднестатистическое по представительной выборке программ множество операций присваивания значений элементам массивов и простым переменным, вычисление значений арифметических выражений, включающих тригонометрические функции с фиксированной и плавающей точкой, вызовы подпрограмм с передачей управления и т. д.

Указанные операции соответствуют операторам языка высокого уровня (типа Фортран), поэтому в научно-технической литературе показатель производительности на программе Whetstone часто интерпретируется как производительность в операторах языка высокого уровня.

Совершенство системы команд СМ1700 подтверждается тем фактом, что число машинных команд только в 2 раза превышает число операторов Whetstone на Фортране (один оператор в среднем интерпретируется двумя командами). «Синтетический» характер программы Whetstone доказывается тем, что показатели производительности для программ на разных языках (Фортран, ПЛ/1) различаются только на несколько процентов и, следовательно, оптимизация при трансляции не сказывается на времени исполнения программы. Для исключения затрат времени на вспомогательные операции (ввод-вывод, обмен и т. п.) обычно программа Whetstone исполняется два раза (100 циклов — t_{100} и 200 циклов — t_{200}). Разность $\Delta T = t_{200} - t_{100}$ определяет «чистое время» на выполнение 100 циклов. Если учесть, что один цикл включает 10^4 операторов Whetstone, то производительность в операторах Whetstone в этом случае определяется по формуле $P_{\text{в}} = 10^6/\Delta T$. Причем в соответствии с ГОСТ 16325—76, п. 2.7б для ЭВМ, допускающих работу с операндами разной длины,

производительность определяется раздельно для операндов одинарной (P_{W1}) и двойной (P_{W2}) длины, а средняя производительность при этом ($P_{W\,ср}$) оценивается в соответствии с формулой

$$P_{W\,ср}=0{,}2P_{W1}+0{,}8P_{W2}.$$

Анализ сравнительных характеристик производительности ЭВМ. Для сопоставления производительности различных ЭВМ, включая различие классов и архитектур, целесообразно определить относительную

$$П^{W}_{отн}=P_{W\,ср}/P_{б\,ср}$$

и эквивалентную

$$П^{W}_{экв}=П^{W}_{отн}П^{к}_{б}$$

производительности, где $P_{W\,ср}$, $P_{б\,ср}$ — средние производительности на программе Whetstone соответственно сравниваемой ЭВМ и ЭВМ, принятой за базовую (например, СМ1420.02 с процессором СМ2420); $П^{к}_{б}=1$ млн коротких команд/с — производительность базовой ЭВМ СМ1420 в коротких командах/с.

Относительная производительность $П^{W}_{отн}$ показывает, во сколько раз производительность сравниваемой ЭВМ отличается от производительности базовой ЭВМ. Эквивалентная производительность $П^{W}_{экв}$ определяет, до какого уровня следует изменить производительность базовой ЭВМ (например, СМ1420.02) в коротких операциях при сохранении ее архитектуры, чтобы достигнуть на программе Whetstone производительности рассматриваемой модели (например, СМ1700). В табл. 8.3 приведены результаты определения абсолютной (для одинарной, двойной, а для ряда моделей — четверной G и восьмерной H точности), средней, относительной и эквивалентной производительности на программе Whetstone. В этой таблице для ряда моделей указано быстродействие в коротких командах в секунду и производительность на смеси Гибсона ($P_{Ги}$) для научно-технических расчетов в соответствии с ГОСТ 16325—76. Одновременно там же приведены данные $T_{Г}$ в секундах, определяющие время обработки типовой программы ГАММ, которая соответствует более сложным научно-техническим расчетам, так как включает преобразование матриц и векторные вычисления.

Из приведенных материалов следует, что чем сложнее нагрузка, тем в большей степени сказывается совершенство архитектуры в оценке производительности. Так, сравнивая СМ1700 и СМ1420, можно отметить, что время выполнения коротких команд у них практически совпадает. На смеси Гибсона для научно-технических расчетов СМ1700 в 1,4 раза производительнее СМ1420. На программе Whetstone преимущество СМ1700 возрастает до 3,06, а на программе ГАММ до 4,7 раз.

В соответствии с этим важным для практики выводом можно по уровню отношения $K_{а}=P_{В}/P_{Ги}$ (производительности при

Таблица 8.5. **Соответствие максимального числа пользователей объему ОЗУ.**

Объем ОЗУ, Мбайт	Максимальное число пользователей
1	12
2	16
3	20

сложной нагрузке $P_{В}$ к производительности при простой нагрузке $P_{Гн}$) оценивать совершенство архитектурных решений семейства ЭВМ. Например, $K_{а СМ1420} = 90/175 = 0,515$, $K_{а ЕС1045} = 0,98$; $K_{а СМ1700} = 1,38$.

Нагрузочные характеристики УВК с архитектурой СМ1700 в многотерминальных конфигурациях. Учитывая особенности МОС ВП, которая в первую очередь предоставляет ресурсы интерактивным запросам, можно отметить, что вариации объемов ОЗУ не приводят к принципиальному изменению производительности ВС в данном режиме. Вместе с этим объем ОЗУ в большей мере сказывается на среднем времени реакции системы на запросы операторов и на пропускную способность в пакетном режиме.

Максимальное число терминалов, которое может быть рекомендовано для использования с ОЗУ емкостью 1—3 Мбайт при достаточно квалифицированной комплексной работе операторов одновременно, приведено в табл. 8.5.

Указанные материалы позволяют уточнить запасы по производительности для случая комплексной загрузки ВС различными по характеру заданиями.

Характеристики режима совместимости СМ1700 и СМ1420 определяются путем прогона одних и тех же программ, написанных на Фортране в режиме совместимости (РС) СМ1700 с СМ1420 и в режиме базовой (РБ) машины. Соотношение производительности в этих режимах СМ1700 зависит как от степени совпадения систем команд СМ1700 и СМ1420, так и от дополнительных системных затрат, включая эмуляцию. В соответствии с этим производительность СМ1700 на вычислительных программах целочисленной арифметики в режиме совместимости составляет $47 \pm 13\%$ от производительности в режиме базовой машины. Соответствующее соотношение производительности СМ1700 на программах действительной арифметики с плавающей точкой двойной точности на Фортране (режима РС к режиму РБ) составляет $21 \pm 1\%$. С учетом соотношения производительности процессоров СМ1700 и СМ1420, примерно равной 3, можно подчеркнуть, что производительность на вычислениях по одним и тем же программам на СМ1700 в режиме совместимости с СМ1420 будет составлять:

для целочисленной арифметики 150%;

для действительной плавающей арифметики 63%.

С учетом потерь на эмуляцию системных функций ОС РВ в МОС ВП для предварительной оценки эффективности режима совместимости СМ1700 можно принять, что в этом режиме

СМ1700 в среднем обеспечивает 60—80% производительности СМ1420.

Показатели производительности СМ1700 в режиме обработки информации в реальном времени. В зависимости от режима работы СВТ, характера решаемых задач при обработке информации в реальном времени пользователей обычно интересуют следующие показатели производительности:

$V_п$ — пропускная способность — объем данных, которые проходят через систему в единицу времени; $\tau_с$ — реакция системы — время, необходимое для идентификации отдельного события и выполнения необходимых ответных действий; $Ф$ — функциональная полнота выполняемых действий — мера алгоритмической сложности процессов принятия решения, реализуемой системой.

Обычно указанные показатели находятся в противоречии. Для реализации предельной пропускной способности необходимо максимально упрощать алгоритмы обработки информации. Этот показатель важен для систем обработки сигналов, измерений в непрерывных режимах.

Повышение скорости реакции требует совершенствования системы прерываний и сокращения затрат времени обработки информации по ряду типовых алгоритмов. Этот показатель важен прежде всего для управления на нижнем уровне высокоскоростными и многокоординатными экспериментальными установками или технологическими процессами.

Высокий уровень функциональности требуется на (2—4)-м уровнях управления. Выполнение комплексных показателей производительности в режиме реального времени СМ1700 потребовало относительно больших аппаратурных затрат на реализацию сложной многоуровневой системы ввода-вывода.

Характеристики времени реакции $\tau_с$, мкс, для СМ1700 (МОС ВП) и СМ1420 (ОС РВ) приведены в табл. 8.6. Причем рассматривались два вида прерываний — от таймера (Т) и обращения к диску (Д).

Для таймерной нагрузки времена реакции моделей СМ1700 и СМ1420 практически совпадают. Для дисковой нагрузки минимальное время реакции СМ1700 и СМ1420 находится практически на одном уровне (минимальное время реакции для СМ1700 на 15% меньше, чем для СМ1420). Среднее время реакции для СМ1700 примерно в 2 раза превышает время реакции для СМ1420. В основном это определяется теми обстоятельствами, что в СМ1700 реализуется большее число нагрузок, большее число типов прерываний (наложение блокирующих и опережающих событий).

В качестве меры пропускной способности системы ($V_п$) можно рассмотреть уровень устойчивой скорости перезаписи информации с диска на ленту, что для СМ1700 соответствует скорости ленты в стартстопном режиме 0,635 м/с. При плотности 63,5 бит/мм $V_п = 63{,}5 \cdot 0{,}635 \cdot 10^3$ бит/с = 40 Кбод.

Таблица 8.6. **Характеристики времени реакции τ_c**

Вид нагрузки	Уровень	τ_c, мкс		Вид нагрузки	Уровень	τ_c, мкс	
		СМ1700	СМ1420			СМ1700	СМ1420
Т	Минимальный	54,0	58,0	Д	Минимальный	51,0	58,0
	Средний	61,7	59,4		Средний	111,5	63,4

Результаты экспериментального исследования затрат процессорного времени на ввод-вывод 4 Кбайт информации в МОС ВП приведены на рис. 8.15. При этом моделировался как форматированный ($\bar{T}_ф(N)$), так и неформатированный ($\bar{T}_н(N)$) режимы ввода-вывода (рис. 8.15, *а*). При фиксированном объеме вводимой/выводимой информации $V=4$ Кбайт, число записей $N=V/N_1$, где N — объем одной записи.

С точностью до коэффициентов вариации σ_x/x, в нашем случае соответственно $\sigma_{T_ф}/\bar{T}_ф$ и $\sigma_{T_н}/\bar{T}_н$, где $\bar{T}_ф$ и $\bar{T}_н$ — среднее время (в секундах) форматированного и неформатированного ввода-вывода (рис. 8.15, *б*), указанные зависимости можно аппроксимировать при $N_1=50—500$:

$$\bar{T}_н(N)=0{,}28\cdot 10^{-3}N_1,$$
$$\bar{T}_ф(N)=0{,}28\cdot 10^{-3}N_1+1{,}1.$$

В связи с тем, что при обмене информацией с использованием программных сетевых средств накладные расходы процессора не превышают 5%, пропускная способность СМ1700 определяется пропускной способностью самих сетевых средств. Для размера буфера 128, 256 и 512 слов на передачу одного сообщения затрачивается примерно 25 нс. Пропускная способность при этом не может превышать 19,2 Кбод (ограничение сетевых средств).

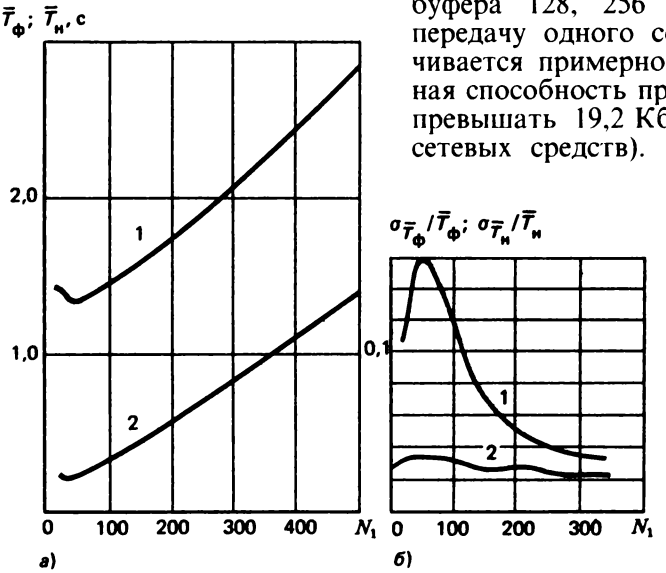


Рис. 8.15. Характеристики времени выполнения ввода-вывода

СПИСОК ИНСТРУКЦИЙ СМ 1700

В приведенном ниже перечне мнемоника некоторых инструкций содержит символ подчеркивания (_). Это означает, что данная инструкция имеет несколько модификаций в зависимости от типа данных, типа перехода, типа доступа, режима работы процессора и состояния отдельных битов. При программировании в мнемонике конкретной инструкции символ подчеркивания заменяется на одну из следующих букв:

тип данных: B - байт
W - слово (по умолчанию)
L - двойное слово (длинное слово)
Q - учетверенное слово (квадрослово)
O - увосьмеренное слово (октослово)
F - одинарный формат (F-формат)
D - двойной формат (D-формат)
G - двойной формат расширенного диапазона (G-формат)
H - учетверенный формат (H-формат)

режим процессора: K - ядро
E - исполнитель
S - супервизор
U - пользователь

состояние бита: C - сброшен
S - установлен

тип перехода: <пусто> - знаковый
U - беззнаковый

тип доступа: R - чтение
W - запись

Логические инструкции для чисел с фиксированной и плавающей запятой

1. MOV_ Пересылка (B,W,L,F,D,H,G,O,Q)
2. MNEG_ Пересылка с отрицанием (B,W,L,F,D,G,H)
3. MCOM_ Пересылка с инверсией (B,W,L)
4. MOVZ_ Пересылка с лидирующими нулями (BW,BL,WL)
5. CLR_ Очистка (B,W,L=F,Q=D=G,O=H)
6. CVT__ Преобразование (B,W,L,F,D,G,H) в (B,W,L,F, D,G,H); кроме BB,WW,LL,FF,HH,DG,GD
7. CVTR_L Преобразование (F,D,G,H) в длинное слово (с округлением)
8. CMP_ Сравнение (B,W,L,F,D,G,H)
9. TST_ Тест (B,W,L,F,D,G,H)
10. BIS_2 Установ битов (B,W,L); 2 операнда
11. BIS_3 Установ битов (B,W,L); 3 операнда
12. BIC_2 Сброс битов (B,W,L); 2 операнда
13. BIC_3 Сброс битов (B,W,L); 3 операнда

14.BIT_ Тест битов (B,W,L)
15.XOR_2 Исключающее "ИЛИ" (B,W,L); 2 операнда
16.XOR_3 Исключающее "ИЛИ" (B,W,L); 3 операнда
17.ROTL Ротация длинного слова
18.PUSHL Загрузка длинного слова в стек

Арифметические инструкции для чисел с фиксированной и плавающей запятой

19.INC_ Инкремент (B,W,L)
20.DEC_ Декремент (B,W,L)
21.ASH_ Арифметический сдвиг (L,Q)
22.ADD_2 Сложение (B,W,L,F,D,G,H); 2 операнда
23.ADD_3 Сложение (B,W,L,F,D,G,H); 3 операнда
24.ADWC Сложение с переносом
25.ADAWI Сложение выравненных слов (с блокировкой памяти)
26.SUB_2 Вычитание (B,W,L,F,D,G,H); 2 операнда
27.SUB_3 Вычитание (B,W,L,F,D,G,H); 3 операнда
28.SBWC Вычитание с переносом
29.MUL_2 Умножение (B,W,L,F,D,G,H); 2 операнда
30.MUL_3 Умножение (B,W,L,F,D,G,H); 3 операнда
31.DIV_2 Деление (B,W,L,F,D,G,H); 2 операнда
32.DIV_3 Деление (B,W,L,F,D,G,H); 3 операнда
33.EMUL Расширенное умножение
34.EDIV Расширенное деление
35.EMOD_ Расширенное умножение (F,D,G,H) с с выделением целой части
36.POLY_ Полином (F,D,G,H)
37.INDEX Индекс массива

Инструкции работы с адресами

38.MOVA_ Пересылка адреса (B,W,L=F,Q=D=G,O=H)-данных
39.PUSHA_ Загрузка в стек адреса (B,W,L=F,D=Q=G,O=H)-данных

Инструкции работы с регистрами процессора

40.PUSHR Загрузка регистров в стек
41.POPR Извлечение регистров из стека
42.MOVPSL Пересылка расширенного слова состояния процессора
43.BI_PSW (C-сброс, S-установ) разрядов в слове состояния программы

Инструкции переходов (безусловные и по кодам условий)

44.BR_ Переход с (B,W)-смещением
45.JMP Универсальный переход
46.BLSS_ Переход (знаковый, U-беззнаковый) по "меньше"
47.BLEQ_ Переход (знаковый, U-беззнаковый) по "меньше или равно"
48.BEQL_ Переход (знаковый, U-беззнаковый) по равенству
49.BNEQ_ Переход (знаковый, U-беззнаковый) по "неравенству"
50.BGEQ_ Переход (знаковый, U-беззнаковый) по "больше или равно"
51.BGTR_ Переход (знаковый, U-беззнаковый) по "больше"
52.BV_ Переход по биту переполнения (C-сброшенному, S-установленному)
53.BC_ Переход по биту переноса (C-сброшенному, S-установленному)

Инструкции сложных переходов

54.ACB_	Сложение, сравнение (B,W,L,F,D,G,H) и переход
55.AOBLEQ	Прибавить единицу и перейти, если "меньше или равно"
56.AOBLSS	Прибавить единицу и перейти, если "меньше"
57.SOBGEQ	Вычесть единицу и перейти, если "больше или равно"
58.SOBGTR	Вычесть единицу и перейти, если "больше"
59.CASE_	Выбор (B,W,L)

Инструкции работы с подпрограммами и процедурами

60.BSB_	Переход к подпрограмме с (B,W)-смещением
61.JSB	Универсальный переход к подпрограмме
62.RSB	Возврат из подпрограммы
63.CALLG	Вызов процедуры из общего вида
64.CALLS	Вызов из процедуры стекового вида
65.RET	Возврат из процедуры

Инструкции для упакованных десятичных строк

66.MOVP	Пересылка
67.CMPP3	Сравнение (3 операнда)
68.CMPP4	Сравнение (4 операнда)
69.ASHP	Арифметический сдвиг с округлением
70.ADDP4	Сложение (4 операнда)
71.ADDP6	Сложение (6 операндов)
72.SUBP4	Вычитание (4 операнда)
73.SUBP6	Вычитание (6 операндов)
74.MULP	Умножение
75.DIVP	Деление
76.CVTLP	Преобразование длинного слова в упакованную строку
77.CVTPL	Преобразование упакованной строки в длинное слово
78.CVTPT	Преобразование строки упакованной в числовую с завершающим знаком
79.CVTTP	Преобразование строки числовой с завершающим знаком в упакованную
80.CVTPS	Преобразование строки упакованной в числовую с лидирующим знаком
81.CVTSP	Преобразование строки числовой с лидирующим знаком в упакованную
82.EDITPC	Редактирование

Инструкции для символьных строк

83.MOVC3	Пересылка (3 операнда)
84.MOVC5	Пересылка (5 операндов)
85.MOVTC	Пересылка с перекодировкой
86.MOVTUC	Пересылка с перекодировкой до помеченного символа
87.CMPC3	Сравнение (3 операнда)
88.CMPC5	Сравнение (5 операндов)
89.LOCC	Локализация символа
90.SKPC	Пропуск символов
91.SPANC	Поиск соответствия
92.SCANC	Поиск несоответствия
93.MATCHC	Поиск подстроки

Инструкции для битовых полей переменной длины

94.	EXTV	Чтение (извлечение) поля
95.	EXTZV	Чтение поля с расширением лидирующими нулями
96.	INSV	Запись (вставка) поля
97.	CMPV	Сравнение поля
98.	CMPZV	Сравнение поля с лидирующими нулями
99.	FF_	Поиск первого (C-сброшенного, S-установленного) бита
100.	BLB_	Переход, если младший бит (C-сброшен, S-установлен)
101.	BB_	Переход, если бит (C-сброшен, S-установлен)
102.	BB__	Переход, если бит (C сброшен, S-установлен), и безусловно (C-сбросить, S-установить) бит
103.	BBCCI	Переход, если бит сброшен, и безусловно сбросить бит (с блокировкой памяти)
104.	BBSSI	Переход, если бит установлен, и безусловно установить бит (с блокировкой памяти)

Инструкции для очередей

105.	INSQUE	Вставить в очередь
106.	INSQHI	Вставить в начало очереди (с блокировкой памяти)
107.	INSQTI	Вставить в конец очереди (с блокировкой памяти)
108.	REMQUE	Удалить из очереди
109.	REMQHI	Удалить из начала очереди (с блокировкой памяти)
110.	REMQTI	Удалить из конца очереди (с блокировкой памяти)

Инструкции специального назначения

111.	CHM_	Смена текущего режима на (K,E,S,U)-режим
112.	PROBE_	Проверка допустимости (R-чтения, W-записи)
113.	SVPCTX	Сохранить контекст процесса
114.	LDPCTX	Загрузить контекст процесса
115.	MTPR	Запись в привилегированный регистр
116.	MFPR	Чтение из привилегированного регистра
117.	REI	Возврат из прерывания
118.	CRC	Циклический контроль
119.	BPT	Ловушка отладчика
120.	XFC	Вызов специальной (микропрограммной) функции
121.	NOP	Отсутствие операции
122.	HALT	Останов

Таким образом, согласно принятой классификации имеется 122 типа инструкций, которые имеют 304 кода операции и 317 мнемоник. (Для удобства программирования некоторым одинаковым кодам операции присвоено несколько мнемоник.) Сводная таблица мнемоник и кодов операций приведена в приложении 2.

Таблица П.2.1. **Однобайтовые коды операций ML**

ML	0	1	2	3	4	5	6	7
0	HALT	NOP	REI	BPT	RET	RSB	LDPCTX	SVPCTX
1	BSBB	BRB	BNEQ	BEQL	BGTR	BLEQ	JSB	JMP
			BNEQU	BEQLU				
2	ADDP4	ADDP6	SUBP4	SUBP6	CVTPT	MULP	CVTTP	DIVP
3	BSBW	BRW	CVTWL	CVTWB	MOVP	CMPP3	CVTPL	CMPP4
4	ADDF2	ADDF3	SUBF2	SUBF3	MULF2	MULF3	DIVF2	DIVF3
5	MOVF	CMPF	MNEGF	TSTF	EMODF	POLYF	CVTFD	
6	ADDD2	ADDD3	SUBD2	SUBD3	MULD2	MULD3	DIVD2	DIVD3
7	MOVD	CMPD	MNEGD	TSTD	EMODD	POLYD	CVTDF	
8	ADDB2	ADDB3	SUBB2	SUBB3	MULB2	MULB3	DIVB2	DIVB3
9	MOVB	CMPB	MCOMB	BITB	CLRB	TSTB	INCB	DECB
A	ADDW2	ADDW3	SUBW2	SUBW3	MULW2	MULW3	DIVW2	DIVW3
B	MOVW	CMPW	MCOMW	BITW	CLRW	TSTW	INCW	DECW
C	ADDL2	ADDL3	SUBL2	SUBL3	MULL2	MULL3	DIVL2	DIVL3
D	MOVL	CMPL	MCOML	BITL	CLRL	TSTL	INCL	DECL
D					CLRF			
E	BBS	BBC	BBSS	BBCS	BBSC	BBCC	BBSSI	BBCCI
F	INSV	ACBL	AOBLSS	AOBLEQ	SOBGEQ	SOBGTR	CVTLB	CVTLW

ML	8	9	A	B	C	D	E	F
0	CVTPS	CVTSP	INDEX	CRC	PROBER	PROBEW	INSQUE	REMQUE
1	BGEQ	BLSS	BGTRU	BLEQU	BVC	BVS	BGEQU	BLSSU
1							BCC	BCS
2	MOVC3	CMPC3	SCANC	SPANC	MOVC5	CMPC5	MOVTC	MOVTUC
3	EDITPC	MATCHC	LOCC	SKPC	MOVZWL	ACBW	MOVAW	PUSHAW
4	CVTFB	CVTFW	CVTFL	CVTRFL	CVTBF	CVTWF	CVTLF	ACBF
5	ADAWI				INSQHI	INSQTI	REMQHI	REMQTI
6	CVTDB	CVTDW	CVTDL	CVTRDL	CVTBD	CVTWD	CVTLD	ACBD
7	ASHL	ASHQ	EMUL	EDIV	CLRQ	MOVQ	MOVAQ	PUSHAQ
7					CLRD		MOVAD	PUSHAD
8	BISB2	BISB3	BICB2	BICB3	XORB2	XORB3	MNEGB	CASEB
9	CVTBL	CVTBW	MOVZBL	MOVZBW	ROTL	ACBB	MOVAB	PUSHAB
A	BISW2	BISW3	BICW2	BICW3	XORWK	XORW3	MNEGW	CASEW
B	BISPSW	BICPSW	POPR	PUSHR	CHMR	CHME	CHMS	CHMU
C	BISL2	BISL3	BICL2	BICL3	XORL2	XORL3	MNEGL	CASEL
D	ADWC	SBWC	MTPR	MFPR	MOVPSL	PUSHL	MOVAL	PUSHAL
D							MOVAF	PUSHAF
E	BLBS	BLBC	FFS	FFC	CMPV	CMPZV	EXTV	EXTZV
F	ASHP	CVTLP	CALLG	CALLS	XFC			

Таблица П.2.2. **Двухбайтовые коды операций MLFD**

M \ L	0	1	2	3	4	5	6	7
0								
1								
2								
3			CVTDH	CVTGH				
4	ADDG2	ADDG3	SUBG2	SUBG3	MULG2	MULG3	DIVG2	DIVG3
5	MOVG	CMPG	MNEGG	TSTG	EMODG	POLYG	CVTGH	
6	ADDH2	ADDH3	SUBH2	SUBH3	MULH2	MULH3	DIVH2	DIVH3
7	MOVH	CMPH	MNEGH	TSTH	EMODH	POLYH	CVTHG	
7								
8								
9								
A								
B								
C								
D								
E								
F							CVTHF	CVTHD

M \ L	8	9	A	B	C	D	E	F
0								
1								
2								
3								
4	CVTGB	CVTGW	CVTGL	CVTRGL	CVTBG	CVTWG	CVTLG	ACBG
5								
6	CVTHB	CVTHW	CVTHL	CVTRHL	CVTBH	CVTWH	CVTLH	ACBH
7					CLRH		MOVAH	PUSHAH
7					CLRO	MOVO	MOVAO	PUSHAO
8								
9	CVTFH	CVTFG						
A								
B								
C								
D								
E								
F								

Примечание. M — младшая тетрада кода операции; L — старшая тетрада кода операции; * — в двухбайтовых кодах операций 1-й байт всегда имеет значение FD.

Базисная 8-битовая таблица

L / M	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ПУС	АР1	<sp>	0	@	P	`	p				А	Р	а	р	Ё
1	НЗ	СУ1	!	1	A	Q	a	q				Б	С	б	с	ё
2	НТ	СУ2	"	2	B	R	b	r				В	Т	в	т	
3	КТ	СУ3	#	3	C	S	c	s				Г	У	г	у	
4	КП	СТП	$	4	D	T	d	t				Д	Ф	д	ф	
5	КТМ	НЕТ	%	5	E	U	e	u				Е	Х	е	х	
6	ДА	СИН	@	6	F	V	f	v				Ж	Ц	ж	ц	
7	ЗБ	КБ	'	7	G	W	g	w				З	Ч	з	ч	
8	ВШ	АН	(	8	H	X	h	x				И	Ш	и	ш	
9	ГТ	КН	)	9	I	Y	i	y				Й	Щ	й	щ	
A	ПС	ЗМ	*	:	J	Z	j	z				К	Ъ	к	ъ	
B	ВТ	АР2	+	;	K	[	k	{				Л	Ы	л	ы	
C	ПФ	РФ	,	<	L	\	l	|				М	Ь	м	ь	
D	ВК	РГ	-	=	M	]	m	}				Н	Э	н	э	
E	ВЫХ	РЭ	.	>	N	^	n	~				О	Ю	о	ю	
F	ВХ	РЭ	/	?	O	_	o	ЗБ				П	Я	п	я	

M - младшая тетрада кода
L - старшая тетрада кода

```
                ПРИМЕР РАБОТЫ С ОЧЕРЕДЯМИ

        .TITLE  МОДУЛЬ ИНИЦИАЛИЗАЦИИ И ОБРАБОТКИ ДАННЫХ
        .SBTTL - ОБЩИЕ ДАНЫЕ, РАЗДЕЛЯЕМЫЕ ДВУМЯ ПРОЦЕССАМИ
;
;Буферы данных первоначально выделяются из разделяемого пула
;Считаем,что процедура ALLOC (рис.4.11) может быть использована
;для их размещения. Переменные, приведенные ниже, описывают
;число и размер буферов.
;       Заметим, что удвоенные символы ** и :: используются
;       для определения переменных, которые являются глобальными

        BUFSIZ==512             ;Размер выделяемого буфера
        NUMBUFS==10             ;Число буферов
;
;Ниже приведены разделяемые переменные.
;Три текущих указателя описывают буфер по мере заполнения
;его данными
;
CURBUF::
        .BLKL   1               ;Адрес текущего буфера
CUREND::
        .BLKL   1               ;Адрес конца буфера
CURPTR::
        .BLKL   1               ;Адрес для помещения следующего
                                ;элемента данных в буфер
;
;Далее следуют заголовки очередей пустых и полных буферов
;
FREELST::
        .BLKQ   1               ;Заголовок свободного списка
FULLST::
        .BLKQ   1               ;Заголовок полного списка

        .SBTTL INIT - ПРОЦЕДУРА ИНИЦИАЛИЗАЦИИ
;++
;       Эта процедура выделяет буферы и инициализирует очереди
;       заголовки очередей и указатели
;
;Входные и выходные параметры:
;       FREELST, FULLST, CURBUF, CURPTR, CUREND, BUFSIZ, NUMBUFS
;
INIT:
        .WORD   ^M<R3>          ;Сохранить регистр R3
;
;Инициализировать заголовки очередей.

        MOVAL   FREELST,FREELST ;Инициализировать заголовок
        MOVAL   FREELST,FREELST+4 ;пустой свободной очереди
        MOVAL   FULLST,FULLST   ;Инициализировать заголовок
        MOVAL   FULLST,FULLST+4 ;пустой заполненной очереди
;
;Выделить буферы и поместить их в свободную очередь. Считаем,
;что процедура ALLOC выделяет буфер заданного размера и
;возвращает его адрес в переданном двойном слове
        MOVL    #NUMBUFS,R3     ;Число получаемых буферов
10$:    PUSHAL  -(SP)           ;Выделить место для адреса
                                ;буфера
        PUSHL   #BUFSIZ         ;Параметры для процедуры
        CALLS   #2,ALLOC        ;выделения буфера
```

```
        BLBC    R1,ERROR            ;Проверка ошибочного состояния
        INSQUE  @(SP)+,FREELST      ;Добавить буфер к свободной
                                    ;очереди
        SOBGTR  R3,10$              ;Продолжить до завершения
;
;Извлечь один буфер из очереди и настроить указатели на сбор данных.
;Первый элемент данных запомнить через 8 байт после начала буфера,
;так как первые два слова удвоенной длины используются как связи
;указателей очереди
;
        REMQUE  @FREELST,CURBUF     ;Взять единицу очереди для
                                    ;текущего буфера
        ADDL3   #8,CURBUF,CURPTR    ;Вычислить адрес для запоминания
                                    ;первого элемента данных
        ADDL3   #512,CURBUF,CUREND  ;Запомнить конечный адрес буфера
        RET                         ;Возврат из подпрограммы

        .SBTTL  COLDAT - ПРОЦЕДУРА СБОРА ДАННЫХ

;++
;       Эта процедура собирает 32-разрядные элементы данных и
;       запоминает их в буфере. Когда буфер заполнится, он ставится
;       в очередь к другому процессу для записи (на диск)
;
;Входные и выходные параметры:
;       FREELST, FULLST, CURPTR, CURBUF, CUREND
;--
COLDAT:
        .WORD   ^M<>                ;Регистры не сохраняются
;
;Принимаем, что GETDAT возвращает 32-разрядный элемент данных в R1
;
        CALLS   #0,GETDAT           ;Получить 32-разрядное значение
        MOVL    CURPTR,R0           ;Указатель запоминаемой величины
        MOVL    R1,(R0)+            ;Запомнить данные в буфере
        CMPL    R0,CUREND           ;Буфер заполнен?
        BLSSU   10$                 ;Переход, если нет
;
;Буфер заполнен. Поставить его в конец очереди полных буферов и
;извлечь из пула другой буфер для сбора данных.
;
        INSQUE  CURBUF,@FULLST+4    ;Поставить в конец "полной"
                                    ;очереди
        CALLS   #00,WAKER           ;"Разбудить" процесс записи
                                    ;буфера
        REMQUE  @FREELST,R0         ;Получить следующий свободный
                                    ;буфер
        BVS     NOBUF               ;Ошибка, если нет буфера
        MOVL    R0,CURBUF           ;Сохранить адрес текущего
                                    ;буфера
        MOVAB   8(R0),R0            ;Адрес для следующего значения
        MOVAB   BUFSIZ(R0),CUREND   ;Запомнить адрес конца буфера
10$:    MOVL    R0,CURPTR           ;Запомнить указатель новых
                                    ;данных
        RET                         ;Возврат

        .SBTTL  OUTPUT - ПРОЦЕДУРА ЗАПИСИ БУФЕРА

;++
;       Эта процедура удаляет буферы из очереди и выводит их
;       на диск. После этого она "спит", пока не будет "разбужена"
;       процессом сборки данных
```

```
;
;Входные и выходные параметры:
;       FREELST, FULLST
;...
OUTPUT:
        .WORD   ^M<R3>              ;Сохранить регистр
;
;Следующая ниже часть программы является бесконечным циклом для
;удаления и обработки записей (входных поступлений).
;
10$:    REMQUE  @FULLST,R3          ;Удалить из головы полной
                                    ;очереди
        BVC     20$                 ;Продолжить, если запись
                                    ;получена
        CALLS   #0,SLEEP            ;Иначе идти "спать"
        BRB     10$                 ;Возобновить работу при
                                    ;"пробуждении"
20$:
;
;Считаем,что процедура OUTBUF при обращении к ней записывает
;буфер на диск: OUTBUF(Адрес буфера,длина)
        PUSHL   #BUFSIZ             ;Загрузить в стек аргумент
                                    ;длины
        PUSHAL  (R3)                ;Загрузить в стек адрес буфера
        CALLS   #2,OUTBUF           ;Записать буфер на диск
        INSQUE  (R3),@FREELST+4     ;Поставить в "свободную"
                                    ;очередь
        BRB     10$                 ;Повторить для других буферов
        .END
```

СПИСОК ЛИТЕРАТУРЫ

1. **Дейкстра Э.** Взаимодействие последовательных процессов.—Языки программирования.—М.: Мир, 1972.—409 с.
2. **Кнут Д.** Искусство программирования для ЭВМ. В 3-х т.: Пер. с англ./Под ред. К. И. Бабенко и В. С. Штаркмана—М.: Мир, 1976.—735 с.
3. **СМ ЭВМ:** Комплексирование и применение/Г. А. Егоров, К. В. Песелев, В. В. Родионов и др.; Под ред. Н. Л. Прохорова.—М.: Финансы и статистика, 1986.—304 с.
4. **Бирюков В. В., Рыбаков А. В., Шакула Ю. П.** Введение в систему программирования ОС РВ.—М.: Финансы и статистика, 1986.—192 с.
5. **Сибеста Р.** Структурное программирование на языке ассемблера ЭВМ VAX-11: Пер. с англ.—М.: Мир, 1988.—536 с.
6. **Малые** ЭВМ и их применение/Ю. А. Дедов, М. А. Островский, К. В. Песелев и др.; Под ред. Б. Н. Наумова.—М.: Статистика, 1980.—231 с.
7. **Егоров Г. А., Родионов В. В.** Комплексы технических средств. Средства вычислительной техники. Вычислительный комплекс СМ1700 // Каталог ГСП.—М.: ИНФОРМПРИБОР, 1988.—С. 32.
8. **Многофункциональная** операционная система, поддерживающая виртуальную память для 32-разрядных ЭВМ/Г. А. Егоров, Г. П. Остапенко, Л. Н. Столяр и др. // Программные продукты и системы.—1988.—№ 4.—С. 30—36.
9. **Прохоров Н. Л., Песелев К. В.** Малые ЭВМ. Перспективы развития вычислительной техники. Кн. 5. М.: Высш. шк., 1989.—158 с.

ОГЛАВЛЕНИЕ