ЛЕНИНГРАДСКИЙ ОРДЕНА ЛЕНИНА
И ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени А. А. ЖДАНОВА

А. Н. Балуев

ЭЛЕМЕНТЫ ПРОГРАММИРОВАНИЯ В СИСТЕМЕ ЕС ЭВМ

Допущено Министерством высшего и среднего
специального образования СССР
в качестве учебного пособия для студентов вузов,
обучающихся по специальности «Прикладная математика»

ЛЕНИНГРАД
ИЗДАТЕЛЬСТВО ЛЕНИНГРАДСКОГО УНИВЕРСИТЕТА
1982

Печатается по постановлению
Редакционно-издательского совета
Ленинградского университета

УДК 681.142.2

Балуев А. Н. Элементы программирования в системе ЕС ЭВМ: Учеб. пособие. — Л.: Изд-во Ленингр. ун-та, 1982. 208 с. Ил. — 33, табл. — 11, библиогр. — 18 назв., прилож. — 3.

В пособии, выпускаемом с рекомендательным грифом Минвуза СССР, описывается логическая структура вычислительной системы ЕС ЭВМ, подробно рассматриваются набор машинных операций и правила составления законченных программ и их фрагментов. Излагаются основные сведения о языке ассемблера, стандартные соглашения о структуре программных модулей, рассматриваются назначение и функции основных компонент операционной системы ОС ЕС.

Рассчитано на студентов математико-механических факультетов университетов и вузов, а также на специалистов в области вычислительных машин и программирования.

Рецензенты: *кафедра алгоритмических языков Моск. ун-та* (зав. каф. доц. Н. П. Трифонов), проф. *В. А. Троицкий* (Ленингр. политехн. ин-т)

Б $\frac{1502000000-181}{076\,(02)-82}$ 01—82

ПРЕДИСЛОВИЕ

Предлагаемая книга является учебным пособием по программированию на языке ассемблера ЕС ЭВМ и предназначена для студентов университетов и технических вузов, специализирующихся в области системного программирования.

Язык ассемблера — машинный язык вычислительной системы. Он труден для человека, громоздок и поэтому мало пригоден для написания обычных прикладных программ. Однако для программиста-профессионала знание машинного языка необходимо по крайней мере по двум причинам. Во-первых, ему приходится изучать, переделывать или писать наново такие системные программы, как супервизоры, загрузчики, обработчики прерываний и другие компоненты операционных систем. А это возможно сделать только средствами машинного языка, наиболее полно и эффективно использующего все возможности, предоставляемые аппаратурой. Во-вторых, разработка новых компиляторов, которая обычно ведется на каком-либо инструментальном языке высокого уровня, может лишь в тех случаях привести к эффективному результату, когда авторы ее ясно представляют себе сильные и слабые стороны используемой аппаратуры, т. е. свободно владеют машинным языком той системы, на которой будет исполняться скомпилированная программа.

У читателя книги предполагается знание АЛГОЛ-60 и наличие некоторого опыта в написании и отладке программ на каком-нибудь универсальном алгоритмическом языке.

В книге принят сжатый стиль изложения, не отвлекающий внимание изучающего деталями, не необходимыми для понимания существа дела. Это позволило изложить в сравнительно небольшом объеме материал, который в переводных учебниках, например [6] и [7], занимает гораздо больше места. Читатель, который хочет получить исчерпывающие сведения по затронутым вопросам, должен обращаться к руководствам, указанным в ссылках в соответствующих местах текста.

Содержание пособия сводится к следующему. В главе 1 приведены краткое описание простейшей конфигурации ЕС ЭВМ и основные характеристики ее компонент: оперативной памяти, процессора, каналов и периферийных устройств. Глава 2 содержит минимальный набор сведений, усвоив которые, изучающий может перейти к упражнениям по написанию и отладке на машине достаточно содержательных программ. Этот набор включает элементы языка ассемблера, описание машинных операций с десятичными целыми числами и простейших переходов, а также нескольких макрокоманд, обеспечивающих ввод и вывод данных и исполнение программы в рамках операционной системы ОС. В главе 3 описываются все машинные операции над двоичными числами, рассматриваются оптимальные приемы организации циклов и выборки компонент прямоугольных массивов, а также простейшие средства динамического распределения памяти в программе с блочной структурой. В главе 4 рассматриваются машинные операции над текстами и кодами и стандартные приемы действий с некоторыми видами данных: вычисление логических выражений, действия с последовательностями битов и байтов, редактирование чисел и пр. В главе 5 описываются основные средства операционной системы ОС ЕС ЭВМ, предназначенные для организации программных модулей и компоновки модулей в большие программы различных структур. Глава 6 содержит описание макросредств языка ассемблера. В приложении 1 приведены таблицы внутренних и перфокарточных кодов, команд процессора, ассемблера и макропроцессора, а также макроопределения нескольких специальных макрокоманд, использованных в тексте учебника.

Автор пользуется случаем выразить глубокую благодарность Н. П Трифонову, В. И. Громыко и В. А. Троицкому, внимательно прочитавшим рукопись и внесшим в нее ряд существенных исправлений.

каждый раз не одна, а целая последовательность команд, в которой одни только начинают исполняться, а другие заканчиваются. Несмотря на это, логика процесса такова, как сказано выше. И если операнд-результат предыдущей команды есть, например, код операции следующей, эта последующая команда выполнится в таком виде, в каком она должна быть сформирована предыдущей при окончательном ее завершении.

Если в последовательности команд встретится команда перехода, то применяется иной способ продвижения адреса текущей команды. В случае команды безусловного перехода в соответствующую часть PSW заносится адрес второго операнда команды перехода. В случае команды перехода по некоторому условию текущий адрес либо продвигается на длину этой команды, либо, если условие перехода выполнено, замещается адресом второго операнда.

Мы уже упоминали о том, что в процессе выполнения команды может возникнуть особая ситуация, требующая прерывания текущей программы. Существует пять групп источников подобных ситуаций: обнаружение технической неисправности машины схемами контроля, поступление внешнего сигнала в систему, программная ошибка, сигнал от канала, команда обращения к супервизору. Опишем действия процессора при возникновении такой ситуации на примере программной ошибки.

В машине различаются 15 типов программных ошибок. Примерами могут служить: нарушения защиты памяти, обнаружение кода несуществующей операции, переполнение регистра при сложении двоичных целых чисел, указание адреса, превышающего максимальный в установке, и др. Мы подробно ознакомимся со всеми возможными ошибками при изучении соответствующих команд.

При обнаружении программной ошибки процессор автоматически предпринимает следующие действия:

заносит номер ошибки в поле «код прерывания» регистра PSW;
записывает в поле ILC (Instruction Length Code) длину в полусловах команды, в которой обнаружена ошибка;
записывает в последние 24 разряда PSW адрес следующей команды;
переносит содержимое регистра PSW в двойное слово памяти с адресом 40;
выбирает в регистр PSW содержимое двойного слова с адресом 104.

После этого процессор возобновляет нормальную деятельность под управлением новой информации в PSW, т. е. переходит к выполнению служебной программы обработки ошибок, адрес начала которой указан в конце двойного слова 104. Эта про-

грамма печатает сообщение о месте и причине ошибки, загружает в оперативную память программу другого пользователя и переключает процессор на ее обработку.

Автор программы может блокировать реакцию процессора на четыре программные ошибки, связанные с действиями над числами. Для этого он должен в начале программы с помощью одной из команд группы 6 занести в поле «маска программы» соответствующий четырехбитовый код.

Подобным образом процессор реагирует на особые ситуации остальных четырех групп, с каждой из которых связана своя пара двойных слов в специальной части оперативной памяти.

Каналы и периферийные устройства

В состав каждой конкретной установки ЕС ЭВМ входит один мультиплексный канал, имеющий номер 0, и от одного до шести селекторных каналов с номерами из диапазона 1—6. К мультиплексному каналу подключены устройства ввода — вывода, количество и состав которых могут быть различными. В минимальном варианте они включают устройство чтения с перфокарт, устройство вывода на перфокарты, алфавитно-цифровое печатающее устройство и электрическую пишущую машинку. К селекторным каналам обычно подключаются устройства внешней памяти на магнитных дисках и магнитных лентах.

Основным средством ввода информации в систему служат 80-колонные перфокарты. В каждой колонке перфокарты с помощью перфоратора, обладающего алфавитной клавиатурой, можно изобразить определенной комбинацией отверстий любой из символов алфавита машины — букву, цифру или специальный знак. Процесс чтения одной карты логически сводится к перенесению содержащейся в ней информации в 80 соседних байтов оперативной памяти системы. При этом каждый символ изображается его внутренним кодом — некоторой комбинацией восьми битов соответствующего байта.

Алфативно-цифровое печатающее устройство предназначено для изображения на бумаге типографскими знаками символов алфавита машины, представленных в оперативной памяти их внутренними кодами. Все эти символы и их коды приведены в Приложении 1.

Устройства памяти на магнитной ленте предназначены для копирования и длительного хранения информации из больших участков оперативной памяти. Ширина используемой ленты 22,7 мм. При движении ленты запись или считывание производятся параллельно девятью магнитными головками, соответствующими восьми основным и одному контрольному биту байта

Несколько байтов с соседними адресами могут объединяться в более крупные ячейки, называемые полусловами, словами или двойными словами. Это объединение производится процессором автоматически при исполнении соответствующей такой ячейке команды. Полуслова состоят из двух байтов, слова — из четырех, а двойные слова — из восьми байтов. Адрес первого байта полуслова, слова или двойного слова должен быть обязательно кратен числу соответственно 2, 4 или 8. Он служит адресом ячейки, указываемым в ссылающейся на ячейку машинной команде. Схема объединения байтов в полуслова, слова и двойные слова приведена на рис. 3.

Кроме перечисленных на рис. 3 ячеек фиксированной длины программист может использовать группу соседних байтов в качестве ячейки переменной длины (не превышающей числа 256).

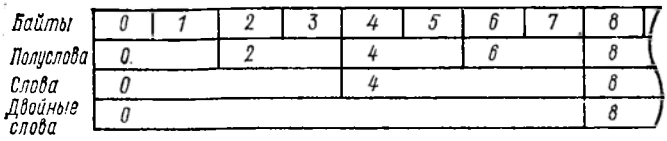


Рис. 3. Схема объединения байтов в ячейки фиксированной длины

Такая ячейка может начинаться с байта с произвольным адресом. Адрес первого байта и их количество указываются в ссылающейся на ячейку команде.

Информация, хранящаяся в ячейках оперативной памяти, по характеру ее использования процессором делится на две группы: команды программы и операнды команд. В машинах серии ЕС нет никакого внутреннего различия в представлении команд и операндов. И если по ошибке программиста процессор воспримет, например, таблицу чисел как последовательность некоторых команд, это может привести к непредсказуемым заранее результатам.

В дальнейшем, говоря об операндах команд, мы будем иногда различать входные операнды (например, слагаемые в команде сложения чисел) и выходные (например, сумму чисел, вырабатываемую командой сложения чисел). Во многих командах входные и выходные операнды могут располагаться в одних и тех же ячейках памяти. Классификация операндов различных команд по смыслу содержащейся в них информации и типы соответствующих им ячеек приведены в табл. 1.

Машины системы ЕС ЭВМ нормально эксплуатируются в мультипрограммном режиме, при котором в различных частях оперативной памяти расположены одновременно несколько не зависящих друг от друга программ. Для того чтобы исключить возможные вследствие ошибок программистов воздействия таких программ друг на друга, в конструкции оперативной па-

Таблица 1

Вид информации	Вид ячейки
Десятичные целые числа в упакованном формате	Ячейки переменной длины от одного до 16 байтов
Десятичные целые числа в формате с зоной	Ячейки переменной длины от 1 до 16 байтов
Двоичные целые числа	Слова и полуслова
Двоичные числа с плавающей точкой	Слова и двойные слова
Тексты	Ячейки переменной длины от 1 до 256 байтов
Коды	Байты, слова, ячейки переменной длины от 1 до 256 байтов

мяти предусмотрены аппаратные средства защиты разделов памяти. Группы соседних байтов размером в 2К объединены в блоки, каждый из которых с помощью специальной команды может быть снабжен четырехбитовым ключом защиты. Так как количество различных ключей составляет 16, вся оперативная память может быть разбита на 16 областей, обладающих различными ключами. Если программа, расположенная в области с ключом n, пытается записать что-то в ячейку из области, защищенной ключом m, выполнение этой программы автоматически прерывается. Область, защищенная ключом 0000, является привилегированной: программа, находящаяся в этой области, может изменять информацию в любой части оперативной памяти. В этой области размещается управляющая программа операционной системы, без услуг которой эксплуатация машины практически невозможна. Привилегированная область всегда включает в себя первые 2К байтов памяти, содержащие ряд ячеек специального назначения. Например, двойные слова с адресами 0, 8 и 16 предназначены для хранения программы начальной загрузки операционной системы.

§ 1.3. Центральный процессор и форматы машинных команд

Машинная порграмма, хранящаяся в оперативной памяти, состоит из последовательности машинных команд и множества операндов. Схема простейшей программы представлена на рис. 4. Предполагается, что адреса ячеек, занимаемых элементами программы, возрастают в направлении слева направо.

Центральный процессор исполняет команды программы последовательно, в порядке их расположения в оперативной памяти (т. е. в порядке возрастания адресов содержащих команды ячеек). Исключения из этого правила возможны только по ко-

мандам переходов, командам переключения состояния и при возникновении особых ситуаций, вызывающих прерывание программы. Поэтому исполнение обычной программы начинается с команды, занимающей ее первую ячейку, и заканчивается исполнением последней команды, вслед за которой располагаются

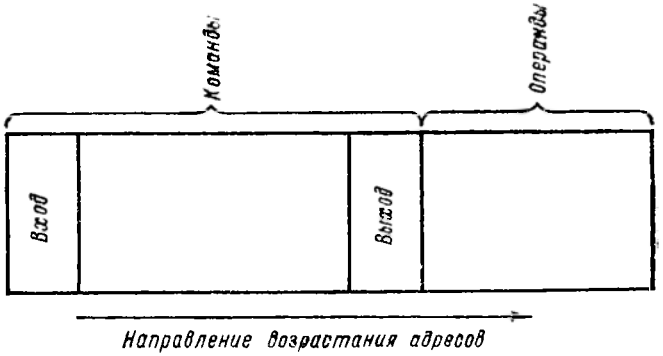


Рис. 4. Схема простой программы

операнды команд. Начало и конец программы, обозначенные на схеме словами «вход» и «выход», имеют определенную специфику, которую мы уточним в следующей главе.

Центральный процессор имеет ряд регистров, предназначенных для хранения некоторых операндов исполняемой программы и различной управляющей информации. В их число входят 16 общих регистров и 4 регистра для операций над числами с плавающей точкой.

Рис. 5. Схема общего регистра

Общий регистр, изображенный на рис. 5, имеет 32 разряда, каждый из которых может хранить один бит. Общие регистры используются для хранения адресов или их слагаемых, для действий над двоичными целыми числами, для обработки текстов и кодов. Каждому регистру присвоен номер из диапазона от 0 до 15.

Регистры для операций над числами с плавающей точкой имеют номера 0, 2, 4 и 6. Емкость каждого из них составляет 64 бита.

Перейдем теперь к изучению структуры машинных команд. Каждая команда программы располагается в ячейке памяти, образованной одним, двумя или тремя соседними полусловами, и имеет две части: поле операции и поле операндов. Поле операции, составляющее всегда первый байт команды, содержит двоичный код, определяющий операцию, выполняемую центральным процессором над операндами команды. В зависимости

от типов операндов и способов указания их адресов все команды разделяются на пять групп (или форматов), обозначаемых на схемах двумя латинскими буквами: RR, RX, RS, SI, SS. Команды формата RR предназначены для операций над двумя регистровыми (Register) операндами. Команды формата RX производят операции типа регистр — память, причем в состав адреса операнда в памяти входит индексное (indeX) слагаемое. Команды формата RS предназначены для операций типа регистр — память (Storage). Команды формата SI производят операции вида память — непосредственный (Immediate) операнд, а команды формата SS — операции типа память — память.

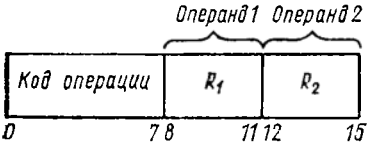


Рис. 6. Схема команды формата RR

Формат RR, занимающий одно полуслово, изображен на рис. 6. Команды этого формата имеют два операнда в двух общих регистрах или в двух регистрах с плавающей точкой. Номера регистров указываются четырехразрядными двоичными числами R_1 и R_2, занимающими левую и правую четверки битов второго байта команды. Результат операции образуется в регистре R_1. Примером этого формата может служить команда сложения двоичных целых чисел, прибавляющая к числу в общем регистре R_1 число из общего регистра R_2.

Формат RX, занимающий два полуслова, изображен на рис. 7. Первый операнд такой команды находится в общем регистре или регистре с плавающей точкой с номером R_1, а второй — в ячейке оперативной памяти, которая может быть (в зависимости от кода операции) байтом, полусловом, словом или двойным словом. Адрес A_2 этой ячейки определяется процессором как сумма трех слагаемых: двоичного числа в общем регистре с номером X_2, двоичного числа в общем регистре с номером B_2 и 12-разрядного двоичного числа D_2, расположенного в последних 12 битах команды. При этом, если сумма выйдет за пределы промежутка от 0 до $2^{24} - 1$, адресом считается двоичное число, изображаемое младшими 24 двоичными разрядами суммы. Коротко это правило можно описать так:

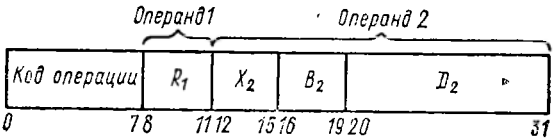


Рис. 7. Схема команды формата RX

$$A_2 = (X_2) + (B_2) + D_2 \bmod 2^{24}.$$

Три слагаемые адреса называются соответственно индексом, базисным адресом и смещением, а регистры X_2 и B_2 — индексным и базисным регистрами команды. В качестве индексного и базисного регистров могут использоваться произвольные общие регистры, за исключением регистра 0. При указании нуля в качестве одного из чисел X_2 или B_2 (или обоих чисел) соответствующие слагаемые не войдут в адрес операнда независимо от того, какое число находится в регистре 0.

На первый взгляд описанный способ указания адреса представляется неоправданно сложным. Но он обладает рядом преимуществ перед непосредственным указанием 24-разрядного адреса в поле операнда. Во-первых, номера регистров X_2 и B_2 и смещение D_2 занимают только 20 битов, что позволяет сделать команду более короткой. Во-вторых, при таком указании адреса программа становится независимой от места ее расположения в памяти. Действительно, если мы сместим программу как единое целое на n байтов, то адреса всех операндов в памяти также изменятся на одно и то же число n, и мы можем компенсировать эту разницу изменением базисного слагаемого адресов в регистре B_2 (или в нескольких регистрах, если в командах программы используется не один базисный регистр). При обработке массивов данных, элементы которых расположены обычно в соседних ячейках, можно, не меняя базисного слагаемого, обращаться в разных командах к различным элементам массива за счет изменения смещения (в пределах от 0 до 4095). Если все элементы массива должны подвергаться одинаковой обработке, можно многократно исполнять одну и ту же группу команд, продвигаясь по массиву за счет изменения индексного слагаемого.

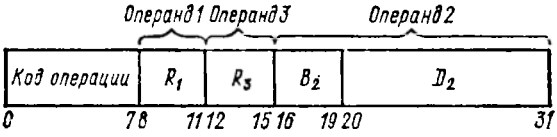


Рис. 8. Схема команды формата RS

Формат RS изображен на рис. 8. R_1 и R_3 означают номера общих регистров, содержащих первый и третий операнды. Второй операнд находится в памяти, и его адрес указывается номером B_2 базисного регистра и смещением D_2. Формирование адреса отличается от случая формата RX только отсутствием индексного слагаемого.

В некоторых командах этого формата часть R_3 поля операндов не используется, а сумма смещения и числа из общего регистра B_2 не служит адресом операнда.

Команда формата SI изображена на рис. 9. Все команды этого формата имеют два операнда. Первый из них расположен

в оперативной памяти и занимает всегда один байт, адрес которого указывается смещением D_1 и номером базисного регистра B_1, так же как в командах формата RS. Второй, непосредственный, операнд I_2 расположен во втором байте ячейки,

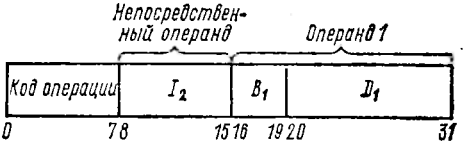


Рис. 9. Схема команды формата SI

занимаемой командой. Примером команды этого формата может служить команда пересылки копии I_2 в байт, адрес которого указывают B_1 и D_1.

Команды формата SS, изображенного на рис. 10, имеют два операнда в оперативной памяти. Команда занимает три полуслова, адреса операндов указываются базисными адресами в регистрах соответственно B_1 и B_2 и смещениями D_1 и D_2. Оба операнда располагаются в ячейках переменной длины. Длина каждого операнда, уменьшенная на 1, указывается 4-разрядными двоичными числами L_1 и L_2. В некоторых командах

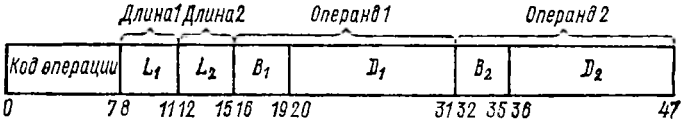


Рис. 10. Схема команды формата SS

этого формата оба операнда имеют одинаковую длину, которая указывается 8-разрядным двоичным числом L. В этих случаях длина операнда может достигать 256 байтов.

По характеру действий, определяемому кодом операции, команды делятся на 6 групп, названия которых и количество операций в группе приведены в табл. 2.

Таблица 2

Название группы	Количество команд
Действия с десятичными целыми числами	9
Действия с двоичными целыми числами	36
Действия с числами с плавающей точкой	44
Логические операции с текстами и кодами	32
Переходы	9
Переключения состояний и ввод — вывод	13

Адрес текущей команды программы и другая информация об условиях ее исполнения хранятся в слове состояния программы в специальном регистре процессора, обозначаемом обычно буквами PSW (Program Status Word). Структура этого регистра и характеристика его содержимого приведены на рис. 11 *.

В поле «ключ», занимающем четыре бита с номерами от 8 до 11, располагается ключ защиты той области памяти, запись в которую разрешена текущей программе. Если при исполнении

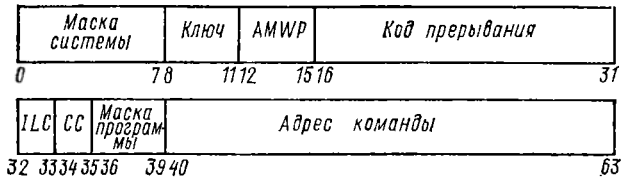


Рис. 11. Схема PSW

какой-нибудь команды процессор обнаруживает, что адрес операнда-результата падает в область, защищенную другим ключом, запись в память не производится и возникает особая ситуация, называемая нарушением защиты памяти.

Единица в бите с номером 14, обозначенном на рис. 11 буквой W, означает, что процессор находится в состоянии ожидания (Waiting). Такая ситуация может возникнуть, например, в том случае, когда программа запросила очередную порцию данных из внешней памяти и не может продолжать работу до их получения. Значение 0 бита W информирует о том, что в рассматриваемый момент процессор занят обработкой команд некоторой программы.

Следующий бит, обозначенный на рисунке буквой P (Program), определяет, в каком из двух возможных взаимно исключающих состояний находится процессор: в состоянии «программа» или в состоянии «супервизор». В состоянии «супервизор» (P = 0) процессор исполняет любую из имеющихся в его ассортименте команд. Если же P = 1, то процессор не имеет права выполнять привилегированные команды — большинство

---

* На самом деле единого регистра не существует, а описываемая ниже информация хранится в разных устройствах процессора. Но так как эта информация в общей сложности составляет 64 бита и при переключениях состояний процессора записывается в (или извлекается из) двойное машинное слово оперативной памяти, удобно говорить о регистре PSW, имеющем структуру двойного слова.

команд, входящих в группу 6. Обнаружение в состоянии P = 1 привилегированной команды считается программной ошибкой.

Бит P защищает операционную систему ЕС ЭВМ от действий неопытного программиста, так как команды группы 6 могут менять управляющую информацию в PSW и обращаться к периферийным устройствам. Но это вовсе не означает, что рядовому пользователю недоступны внешние устройства. Мы уже упоминали о том, что в области памяти, защищенной ключом 0000, находится управляющая программа операционной системы, главную часть которой составляет супервизор — программа, предназначенная для управления всеми привилегированными действиями в системе: запуском и остановкой периферийных устройств, обработкой особых ситуаций. Среди команд группы 6 есть команда обращения к супервизору, которая может использоваться в обычной программе. Она определяет, какая из обширного множества стандартных услуг требуется автору программы. Обнаружив эту команду в программе, исполняемой в состоянии P = 1, процессор автоматически переходит в состояние P = 0 и передает управление супервизору. После того, как последний выполнит заказ, процессор возвращается в прежнее состояние и продолжает выполнение прерванной на время работы супервизора обычной программы.

Биты 34 и 35, обозначенные на рисунке буквами CC, определяют код условия (Condition Code), который вырабатывает некоторые команды в зависимости от вида операнда-результата. Например, команды сложения чисел определяют значение CC по следующему правилу: 00 — сумма равна нулю, 01 — сумма отрицательна, 10 — сумма положительна, 11 — сумма превышает допустимые пределы. Среди команд перехода есть такие, которые нарушают естественный порядок исполнения лишь при наличии определенного значения CC, что дает автору программы разветвлять алгоритм в зависимости от полученных на предыдущих шагах результатов.

Последние 24 бита PSW служат для хранения адреса исполняемой команды. Действия процессора при обработке любой из команд групп 1 — 4 можно разбить на следующие этапы:

выборка очередной команды в служебные регистры;
выборка входных операндов в служебные регистры;
вычисление результатов и выработка кода CC;
запись результатов в память;
продвижение адреса очередной команды в PSW на длину исполненной команды.

На самом деле процесс обработки команд гораздо более сложен. Для ускорения работы машины ее схемы совмещают во времени исполнение текущей команды с предварительной обработкой следующих, и в «поле зрения» процессора находится

каждый раз не одна, а целая последовательность команд, в которой одни только начинают исполняться, а другие заканчиваются. Несмотря на это, логика процесса такова, как сказано выше. И если операнд-результат предыдущей команды есть, например, код операции следующей, эта последующая команда выполнится в таком виде, в каком она должна быть сформирована предыдущей при окончательном ее завершении.

Если в последовательности команд встретится команда перехода, то применяется иной способ продвижения адреса текущей команды. В случае команды безусловного перехода в соответствующую часть PSW заносится адрес второго операнда команды перехода. В случае команды перехода по некоторому условию текущий адрес либо продвигается на длину этой команды, либо, если условие перехода выполнено, замещается адресом второго операнда.

Мы уже упоминали о том, что в процессе выполнения команды может возникнуть особая ситуация, требующая прерывания текущей программы. Существует пять групп источников подобных ситуаций: обнаружение технической неисправности машины схемами контроля, поступление внешнего сигнала в систему, программная ошибка, сигнал от канала, команда обращения к супервизору. Опишем действия процессора при возникновении такой ситуации на примере программной ошибки.

В машине различаются 15 типов программных ошибок. Примерами могут служить: нарушения защиты памяти, обнаружение кода несуществующей операции, переполнение регистра при сложении двоичных целых чисел, указание адреса, превышающего максимальный в установке, и др. Мы подробно ознакомимся со всеми возможными ошибками при изучении соответствующих команд.

При обнаружении программной ошибки процессор автоматически предпринимает следующие действия:

заносит номер ошибки в поле «код прерывания» регистра PSW;
записывает в поле ILC (Instruction Length Code) длину в полусловах команды, в которой обнаружена ошибка;
записывает в последние 24 разряда PSW адрес следующей команды;
переносит содержимое регистра PSW в двойное слово памяти с адресом 40;
выбирает в регистр PSW содержимое двойного слова с адресом 104.

После этого процессор возобновляет нормальную деятельность под управлением новой информации в PSW, т. е. переходит к выполнению служебной программы обработки ошибок, адрес начала которой указан в конце двойного слова 104. Эта про-

грамма печатает сообщение о месте и причине ошибки, загружает в оперативную память программу другого пользователя и переключает процессор на ее обработку.

Автор программы может блокировать реакцию процессора на четыре программные ошибки, связанные с действиями над числами. Для этого он должен в начале программы с помощью одной из команд группы 6 занести в поле «маска программы» соответствующий четырехбитовый код.

Подобным образом процессор реагирует на особые ситуации остальных четырех групп, с каждой из которых связана своя пара двойных слов в специальной части оперативной памяти.

§ 1.4. Каналы и периферийные устройства

В состав каждой конкретной установки ЕС ЭВМ входит один мультиплексный канал, имеющий номер 0, и от одного до шести селекторных каналов с номерами из диапазона 1—6. К мультиплексному каналу подключены устройства ввода — вывода, количество и состав которых могут быть различными. В минимальном варианте они включают устройство чтения с перфокарт, устройство вывода на перфокарты, алфавитно-цифровое печатающее устройство и электрическую пишущую машинку. К селекторным каналам обычно подключаются устройства внешней памяти на магнитных дисках и магнитных лентах.

Основным средством ввода информации в систему служат 80-колонные перфокарты. В каждой колонке перфокарты с помощью перфоратора, обладающего алфавитной клавиатурой, можно изобразить определенной комбинацией отверстий любой из символов алфавита машины — букву, цифру или специальный знак. Процесс чтения одной карты логически сводится к перенесению содержащейся в ней информации в 80 соседних байтов оперативной памяти системы. При этом каждый символ изображается его внутренним кодом — некоторой комбинацией восьми битов соответствующего байта.

Алфативно-цифровое печатающее устройство предназначено для изображения на бумаге типографскими знаками символов алфавита машины, представленных в оперативной памяти их внутренними кодами. Все эти символы и их коды приведены в Приложении 1.

Устройства памяти на магнитной ленте предназначены для копирования и длительного хранения информации из больших участков оперативной памяти. Ширина используемой ленты 22,7 мм. При движении ленты запись или считывание производятся параллельно девятью магнитными головками, соответствующими восьми основным и одному контрольному биту байта

оперативной памяти. Плотность записи может составлять либо 8, либо 32 байта на один погонный миллиметр ленты, а скорость передачи — 64К или 256К в одну секунду. Предельная емкость одной бобины с лентой составляет около 20 млн байтов.

Схема пакета магнитных дисков изображена на рис. 12. Пакет состоит из шести дисков диаметром около полуметра, закрепленных на общей оси, десять рабочих поверхностей которых (исключая две наружных) покрыты магнитным лаком. Гребенка с десятью магнитными головками может занимать одно из 203 фиксированных положений, называемых цилиндрами. Каждый цилиндр состоит из десяти дорожек, по которым

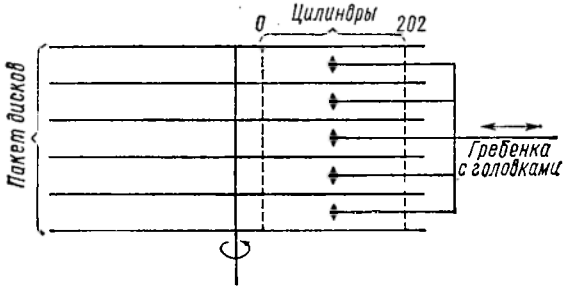


Рис. 12. Схема пакета магнитных дисков

при вращении пакета скользят магнитные головки. В отличие от магнитной ленты запись или считывание информации из оперативной памяти в каждый момент времени могут производиться только по одной из десяти головок. Поэтому девять битов одного байта располагаются на дорожке друг за другом. Емкость одной дорожки составляет 3200 байтов, а всего пакета — 7,2 млн байтов. Скорость передачи — 156К байтов в секунду. Магнитный диск является гораздо более оперативным устройством, чем магнитная лента. Скорости вращения пакета и передвижения гребенки таковы, что время доступа к любому участку записи измеряется тысячными долями секунды.

Управление работой устройств внешней памяти и ввода — вывода производится с помощью особых команд, называемых командами канала. Последовательности таких команд располагаются в привилегированной части оперативной памяти. В отличие от обычных команд они расшифровываются не процессором, а каналами системы. Результатом исполнения каждой команды является определенное действие одного из связанных с каналом устройств. Например, в случае магнитных дисков такими действиями могут быть: передвижение гребенки с головками, перепись информации из указанного в команде уча-

стка оперативной памяти на одну из дорожек, поиск на дорожке диска участка записи с определенными свойствами и т. д. Сигналом к началу работы канала является выполнение процессором одной из привилегированных команд группы 6, указывающих номер канала и номер устройства. Адрес начала программы канала должен быть предварительно занесен во вторую половину двойного слова с адресом 72. Инициировав работу канала и периферийного устройства, процессор может продолжать исполнение своей программы параллельно с действиями канала и устройства. Когда последнее закончит работу канал посылает в управляющие системы процессора сигнал прерывания и сообщает в коде прерывания текущего PSW свой номер и номер устройства.

Написание программы канала, ее запуск и исследование ситуации, возникшей после ее окончания, доступны лишь весьма квалифицированному программисту. Обычно для ввода, вывода и обмена с внешней памятью пользуются одной из стандартных процедур обращения к супервизору, запускаемых специальными макрокомандами (см. § 2.4), определяющими вид услуги и ее параметры.

Глава 2

ЭЛЕМЕНТЫ ЯЗЫКА АССЕМБЛЕРА

§ 2.1. Язык ассемблера

Для дальнейшего изложения нам необходимо воспользоваться системой программирования «ассемблер». Дело в том, что изображать на бумаге будущую машинную программу точно в той форме, в какой она будет храниться в оперативной памяти во время исполнения ее процессором, очень неудобно. Во-первых, нам пришлось бы помнить 8-разрядные двоичные коды (а их около ста сорока!), обозначающие машинные операции. Во-вторых, вместо привычных буквенных обозначений переменных нужно было бы писать двоичные номера базисных и индексных регистров и 12-разрядные смещения, определяющие операнды команд.

Система «ассемблер» состоит из двух основных компонент: языка ассемблера, позволяющего в удобной для человека форме описывать машинную программу, и программы-компилятора, переводящей запись на языке ассемблера в машинную форму. Процедура составления программы с помощью этой системы вкратце сводится к следующему. Сначала программист, пользуясь принятыми в языке средствами, на специальном бланке пишет в нужном порядке команды входной порграммы. Каждая строка бланка имеет 80 позиций, разбитых на пять полей: имя, операция, операнды, комментарии, идентификация, и соответствует одной перфокарте. Поле идентификации предназначено для номера перфокарты и занимает последние восемь позиций с номера 73 до 80. Поле имени начинается всегда с первой позиции. Если имя отсутствует, первая позиция бланка должна содержать пробел. Остальные поля не имеют жестких границ. Они размещаются между полем имени и позицией 72 исключительно, разделяясь одним или несколькими пробелами.

Алфавит языка ассемблера состоит из четырех групп символов: букв, цифр, специальных знаков, дополнительных знаков. Первую группу составляют 28 прописных латинских букв

A, B, C, ..., *Z*

и три особые буквы

$$¤\ \#\ @$$

Вторую группу образуют десять цифр, а третью составляют одиннадцать «видимых» знаков

$$+\ -\ =\ *\ (\ )\ '\ /\ \&\ .\ ,$$

и пробел, который изображается на бланке пустой позицией. К дополнительным знакам относятся все остальные символы ДКОИ (см. Приложение 1), не вошедшие в приведенный список. В частности, они содержат русские буквы, отличные по начертанию от латинских. Дополнительные символы нельзя писать на поле имени или на поле операции. В некоторых случаях их можно употреблять на поле операндов и без ограничений — на поле комментария.

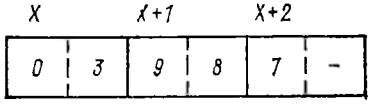
Входная программа может состоять из записей трех типов: машинных команд, команд ассемблера и макрокоманд. Эти записи переносятся на перфокарты, которые вместе с несколькими управляющими картами, содержащими заказ на выполнение компилятора в определенном режиме, вводятся в машину. Компилятор ассемблера перерабатывает входную программу в эквивалентную ей объектную порграмму, которая может быть выдана на перфокарты, помещена во внешнюю память для хранения или передана процессору для исполнения. Компилятор выдает также листинг программы — отпечатанный на алфавитно-цифровом печатающем устройстве документ, содержащий параллельные тексты входной и объектной программ, сообщения об ошибках и некоторые другие сведения.

Детальное описание языка ассемблера читатель может найти в руководствах [5—7].

§ 2.2. Операции с десятичными целыми числами

Мы начнем изучение машинных операций с действий над десятичными числами.

Операндами команд десятичной арифметики служат целые десятичные числа в упакованном формате. Расположение в памяти числа, скажем, —3987 иллюстрирует следующая схема:



В общем случае $2n+1$-разрядное число занимает $n+1$ соседний байт с адресами: X, $X+1$, ..., $X+n$. Каждый байт делится на две части по 4 бита, в которых (в байтах X, $X+1$, ..., $X+n-1$) размещаются в порядке убывания стар-

шинства цифры числа, а в последнем байте — цифра единиц и знак. Двоичные коды цифр и знаков приведены в табл. 3.

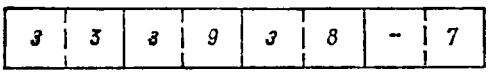
Таблица 3

Цифра	Код цифры	Знак	Код знака
0	0000	+	1010
1	0001	—	1011
2	0010	+	1100
3	0011	—	1101
4	0100	+	1110
5	0101	+	1111
6	0110		
7	0111		
8	1000		
9	1001		

Из табл. 3 видно, что один и тот же знак может иметь различные представления. Но если операнд является результатом, вырабатываемым командами сложения, вычитания, умножения или деления, в качестве кода «плюс» всегда записывается 1100, а в качестве кода «минус» всегда записывается 1101.

Длина операнда может меняться от 1 до 16 байтов, т. е. от 1 до 31 цифры, причем количество цифр всегда нечетное.

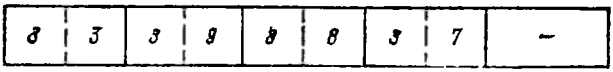
Существует еще одна форма хранения в оперативной памяти десятичного числа, называемая зонным форматом. Она используется для представления числа перед выдачей его на печатающее устройство или после чтения с перфокарты. Имеются специальные команды для перевода упакованного формата в зонный и обратно. Однако для того чтобы напечатать десятичное число, недостаточно перевести его в зонный формат. Например, число —3987 в зонном формате занимает 4 байта:



Буква «з» на схеме означает двоичный код 1111, называемый кодом зоны. Если мы хотим напечатать наше число в виде

3987—

мы должны заменить его последний байт двумя



Здесь «—» означает 8-разрядный двоичный код литеры минус. (Подробнее об этом см. в § 4.4.)

Таблица 4

Название операции	Возможные прерывания	Код условия			
		00	01	10	11
AP	З, А, ПД, Д	0	<0	>0	Переполнение
SP	З, А, ПД, Д	0	<0	>0	Переполнение
ZAP	З, А, ПД, Д	0	<0	>0	Переполнение
CP	А, Д	=	<	>	
MP	З, А, Д, С	Не изменяют код условия			
DP	З, А, Д, С, ДД				
PACK	З, А				
UNPK	З, А				
MVO	З, А				

Рассмотрим теперь команды десятичной арифметики. В левой колонке табл. 4 приведены обозначения кодов команд на языке ассемблера. Все команды имеют формат SS, поле длины каждой команды содержит длины L_1 и L_2 обоих операндов. Буква P в названиях первых шести операций подчеркивает, что оба операнда соответствующих команд — десятичные числа в упакованном (Pack) формате

Операции имеют следующий смысл:

AP — десятичное сложение (Add),
SP — десятичное вычитание (Subtract),
ZAP — сложение с предварительной очисткой (Zero and Add),
CP — десятичное сравнение (Compare),
MP — десятичное умножение (Multiply),
DP — десятичное деление (Divide),
PACK — упаковка (PACK),
UNPK — распаковка (UNPacK),
MVO — пересылка со сдвигом (MoVe with Offset).

Во второй колонке приведены все причины прерывания программы, которые могут произойти при выполнении команды:

З — попытка записи в защищенную область,
А — адрес операнда превышает максимально допустимый,
Д — неправильный код цифры или знака в операнде,
ПД — результат не помещается на отведенное для него поле,
ДД — частное не помещается на отведенное для него поле,
С — неправильная спецификация операндов.

В третьей колонке приведены значения кодов условий, вырабатываемых четырьмя первыми командами. Обозначения имеют следующий смысл:

0 — результат равен нулю,
<0 — результат меньше нуля,

> 0 — результат больше нуля,
$=$ — операнды равны,
$<$ — первый операнд меньше второго,
$>$ — первый операнд больше второго.

Рассмотрим правила указания операндов в командах десятичной арифметики. Каждый операнд можно задавать либо в явном, либо в неявном формате. Явное указание обоих операндов в команде AP описывается следующей схемой:

AP $d_1(l_1, b_1), d_2(l_2, b_2)$

В этой схеме d_i, l_i, b_i обозначают соответственно смещение, длину и номер базисного регистра операнда. Например

AP 0(6, 12), 98(4, 11)

— запись на языке ассемблера команды десятичного сложения. Ее первый операнд имеет длину 6 байтов, а адрес его первого байта хранится в общем регистре 12. Второй операнд имеет длину 4 байта, а его адрес получается добавлением числа 98 к адресу в общем регистре 11.

Заметим, что порядок расположения d_i, l_i и b_i в явном формате отличен от порядка расположения соответствующих элементов в машинном коде. Кроме того, длина l_i в явном формате означает фактическую длину операнда в отличие от машинной длины, которая на единицу меньше фактической.

В нашем примере d_i, l_i и b_i указаны десятичными числами без знаков. В общем случае их можно задавать числами в 16-ричном, двоичном и литерном изображении и их арифметическими комбинациями, а также более сложными формулами, называемыми абсолютными выражениями (см. [5, с. 17, 29]).

Явный формат команды мало отличается от машинного и употребляется в ассемблерной программе в исключительных случаях. Обычно операнды описываются в неявном формате с использованием ассемблерных имен — идентификаторов, обозначающих некоторые машинные адреса. Длина такого идентификатора, состоящего из букв и цифр и начинающегося обязательно с буквы, не должна превышать 8. Структура неявного формата на примере той же команды AP описывается такой схемой:

AP $s_1(l_1), s_2(l_2)$

где l_i имеет тот же смысл, что и в явном формате, а s_i означает переместимое выражение, определяющее адрес операнда. Перемещаемое выражение обычно состоит из ассемблерного имени, к которому могут быть дописаны справа со знаком плюс или минус одно или несколько абсолютных выражений. Например

AP A1(5), B2 + 10(3)

означает команду сложения двух упакованных десятичных чисел. Первое из них имеет длину 5 байтов (т. е. содержит 9

цифр) и начинается с байта с адресом A1. Второе слагаемое имеет длину 3 байта (т. е. пять цифр) и начинается с адреса, большего на 10, чем адрес, обозначенный идентификатором B2.

Абсолютное выражение l_1 и заключающие его скобки могут отсутствовать как в одном, так и в обоих операндах неявного формата. В таких случаях длина операнда определяется по характеристике длины (см. ниже) ассемблерного имени, входящего в выражение s_1.

Компилятор с языка ассемблера, преобразуя входную запись в объектную форму, должен заполнить адресные части машинного кода двоичными значениями смещений и номеров базисных регистров. Чтобы он мог это сделать, автор программы обязан сообщить компилятору, во-первых, какие регистры можно использовать в качестве базисных и какие адреса будут находиться в этих регистрах во время исполнения машинной программы, и, во-вторых, где по отношению к началу программы расположены байты, адреса которых он обозначил идентификаторами A1 и B2. Вопрос о том, как можно указывать базисные регистры и их содержимое, мы рассмотрим несколько позже. Что же касается определения значения ассемблерных имен, это можно сделать с помощью команд ассемблера DC — определение постоянной (Define Constant) и DS — определение памяти (Define Storage). Эти команды пишутся в том месте входной программы, где мы хотим расположить соответствующие операнды (обычно в конце). В нашем случае мы можем определить A1 и B2, например, так:

```
A1      DC      PL5'-192'
B2      DS      3PL5
```

Компилятор ассемблера, преобразуя входную программу в объектную, просматривает ее команды в порядке сверху вниз и точно в таком же порядке вырабатывает соответствующие объектные команды. Как уже говорилось выше, машинная команда входной программы перерабатывается в двоичный код соответствующей команды процессора. Если же очередная команда входной программы — команда ассемблера DC, то компилятор вставляет в соответствующее место объектной программы двоичный код постоянной, описанной на поле операндов. Кроме того, если на поле имени команды расположен некоторый идентификатор, компилятор вносит этот идентификатор в таблицу имен, запоминая при этом расположение соответствующего байта по отношению к началу программы и некоторую дополнительную информацию, о которой мы скажем немного позже. В нашем случае компилятор вырабатывает упакованное (Pack) десятичное число длины (Length) 5 со значением —192 (т. е. перед цифрой 1 будет написано еще шесть нулей) и запоминает, что адрес первого байта числа обозначен идентификатором A1.

Если же очередная строка входной программы — команда ассемблера DS, то компилятор резервирует в соответствующем месте объектной программы поле заказанной длины и при наличии идентификатора вносит его в таблицу, запоминая относительное расположение первого байта этого поля и некоторую дополнительную информацию. При этом на резервированное поле никакой записи не производится, и его состояние зависит от того, что осталось на этом месте от предыдущей программы.

В нашем примере запись на поле операндов команды DS означает заказ трех соседних полей по пять байтов каждое. Так как команды ассемблера с именами A1 и B2 расположены непосредственно друг за другом, ассемблерное имя B2, помечающее первый байт, резервируемый командой DS, имеет то же значение, что и переместимое выражение A1 + 5.

Дополнительная информация, которая запоминается в таблице для каждого идентификатора, включает в себя его характеристику длины. В нашем примере характеристика длины указана явно для обоих идентификаторов числом 5, написанным после буквы L. Эта характеристика автоматически используется компилятором при формировании объектного кода для операнда машинной команды, в котором опущено явное указание длины. Например, запись во входной программе

```
AP      A1, B2+10(3)
```

эквивалентна такой:

```
AP      A1(5), B2+10(3)
```

А если мы опустим указание длины 3 во втором операнде, это будет то же самое, что его запись в виде

$$B2 + 10(5).$$

Подробнее о командах DC и DS см. [5, с. 143].

Рассмотрим теперь правила выполнения команд десятичной арифметики. Мы будем изображать их операнды в виде

$$A(l_1),\ B(l_2),$$

имея в виду, что буквы l_1 и l_2 означают числа из диапазона от 1 до 16. Поля в оперативной памяти, занятые операндами, могут перекрываться, но только так, что их младшие, т. е. самые правые байты совпадают (в противном случае знаковый полубайт одного операнда совпадает с числовым в другом операнде, что вызовет прерывание по причине Д). Начнем с команды

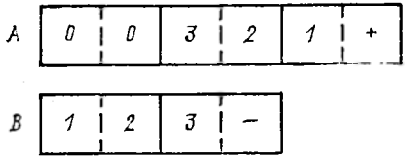
```
AP      A(l1), B(l2)
```

Если мы воспользуемся обозначениями АЛГОЛа и предположим, что A и B означают не только адреса первых байтов операндов, но и значения самих операндов, результат выполнения можно описать так:

$$A := A + B.$$

Более точно происходит следующее. Если хотя бы один операнд содержит ошибку в коде цифры или знака, происходит прерывание программы по причине Д. Если коды правильные, процессор вычисляет сумму чисел A и B и записывает ее в упакованном виде на поле первого операнда. Если мест на этом поле больше, чем значащих цифр в сумме, результат дополняется соответствующим количеством ведущих нулей. Если же сумма не помещается в поле первого операнда, туда записываются знак и $2 \times l_1 - 1$ младших разрядов и возникает прерывание программы по причине ПД.

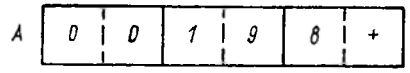
Рассмотрим пример. Пусть значения A и B перед операцией таковы:



После выполнения команды

AP A(3), B(2)

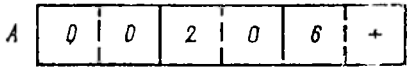
на поле A появится значение



Если после этого выполнить команду

AP A(3), A + 2(1)

на поле A появится



Команда вычитания

SP $A(l_1)$, $B(l_2)$

отличается от AP лишь в том, что знак второго операнда воспринимается как обратный.

Команда сложения с предварительной очисткой

ZAP $A(l_1)$, $B(l_2)$

производит действие

$$A := B.$$

Прерывание ПД происходит, если значение B не помещается на поле A, а прерывание Д в том случае, когда неправилен код

второго операнда. В отличие от двух предыдущих команд допускается и такое перекрытие полей, когда младший байт первого операнда лежит правее младшего байта второго.

Команда сравнения

CP A(l_1), B(l_2)

не меняет значений операндов, а только вырабатывает код условия. Прорывание Д возникает, если хотя бы один из операндов имеет неправильный код или поля перекрываются так, что их правые байты не совпадают.

Команда умножения

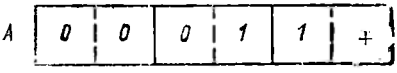
MP A(l_1), B(l_2)

производит действие

$$A := A \times B.$$

Однако для правильного осуществления умножения должны быть выполнены следующие условия. Во-первых, длина второго операнда l_2 должна быть не более 8 и по крайней мере на единицу меньше длины l_1 первого. В противном случае умножение не происходит и возникает прерывание по причине С. Во-вторых, перед первой значащей цифрой первого операнда должно быть не менее чем $2 \times l_2$ нулей. Если это условие не выполнено, умножение не происходит и возникает прерывание программы по причине Д. Прерывание по причине Д происходит также при неправильном коде одного из операндов или неправильном перекрытии их полей. При правильном завершении умножения результат содержит по крайней мере один ведущий ноль.

Пусть, например, поле А длиной 3 содержит число 11:



Хотя $121 = 11^2$ свободно размещается на поле А, команда

MP A(3), A + 1(2)

вызывает прерывание по причине Д: количество ведущих нулей на поле результата до умножения должно быть не менее 4. Дело можно поправить, увеличив предварительно поле А еще на один байт и записав туда нули:

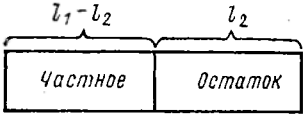
ZAP A — 1(4), A(3)
MP A — 1(4), A + 1(2)

Мы предполагаем здесь, что байт с адресом А — 1 не содержал никакой нужной в дальнейшем информации.

Команда деления

DP A(l_1), B(l_2)

размещает на поле A целую часть A ÷ B и остаток A — A ÷ ÷ B × B от деления прежнего значения на поле A на делитель B. Расположение результата на поле A ясно из схемы:



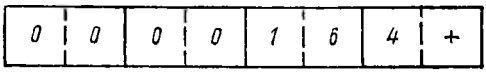
Знак частного определяется по правилу: sign(A) × sign(B), а знак остатка совпадает со знаком делимого.

Команда DP требует соблюдения двух дополнительных условий. Во-первых, длина делителя l_2 не должна превышать 8, а длина делимого l_1 должна быть больше длины l_2 делителя. В противном случае возникает прерывание по причине С. Во-вторых, частное должно поместиться на поле длины $l_1 - l_2$. В противном случае возникает прерывание по причине ДД.

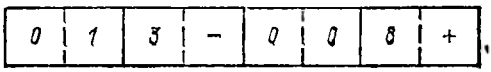
Например, частное и остаток от деления чисел, хранящихся на полях A и B длиной в 2 байта каждое, можно получить на поле REZ длиной в 4 байта так:

```
ZAP     REZ(4), A(2)
DP      REZ(4), B(2)
```

Если делимое равно 164, а делитель есть —12, то после выполнения ZAP на поле REZ будет



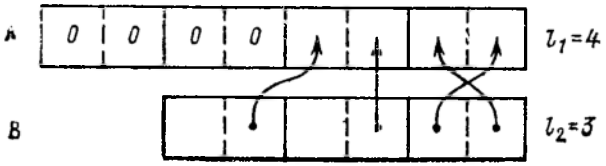
а после DP



Команда упаковки

PACK A(l_1), B(l_2)

преобразует десятичное число в зонном формате, расположенное на поле B, в десятичное упакованное и записывает его на поле A. Действие PACK поясняет схема:

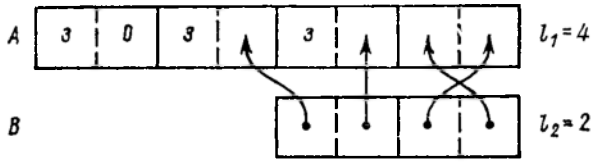


Если в старшей части поля A остаются свободные места, они заполняются кодами упакованной цифры 0. Если же на поле A не хватает места для всех цифр с поля B (т. е. $2xl_1 < l_2 + 1$), старшие цифры числа с поля B теряются. Код операнда на поле B не проверяется, т. е. на поле A может сформироваться результат, не являющийся кодом упакованного десятичного числа.

Команда

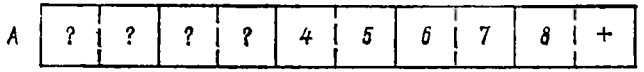
UNPK A(l_1), B(l_2)

производит обратную операцию, преобразуя код упакованного десятичного числа в код того же числа в зонном формате:



Если длина первого операнда недостаточна для размещения всех цифр с поля B (т. е. $l_1 < 2 \times l_2 - 1$), старшие цифры пропадают. Если же $l_1 > 2 \times l_2 - 1$, старшие байты A заполняются кодами цифры 0 в зонном формате.

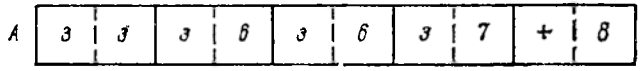
Поля операндов A и B в командах PACK и UNPK могут перекрываться произвольным образом. При выполнении обеих операций процессор считывает по мере необходимости в порядке справо налево байты второго операнда и в таком же порядке побайтно записывает в память результат. Если, например, поле A длины 5 содержало



т. е. операнд A + 2(3) имел значение +45 678 в упакованном формате, то, выполнив команду

UNPK A(5), A + 2(3)

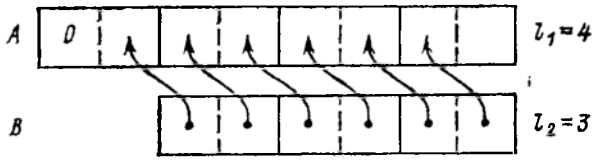
мы получим на A(5) не +45 678 в зонном формате, а



Действие команды

MVO A(l_1), B(l_2)

при $l_1 > l_2$ описывается следующей схемой



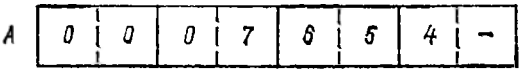
Код, находившийся в правой половине последнего байта первого операнда, не меняется, а «лишние» левые полубайты заполняются кодами упакованной цифры 0. Если же $l_1 \leqslant l_2$, старшие полубайты второго операнда не используются. Поля A и B могут перекрываться произвольным образом. Результат формируется побайтно в порядке справа налево, как в командах PACK и UNPK.

С помощью MVO можно сдвигать вправо код упакованного числа с уничтожением младших разрядов. Если, например, на поле A (4) хранилось — 7 654 321:

то после выполнения команды

```
MVO     A(4), A(2)
```

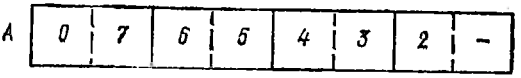
там будет



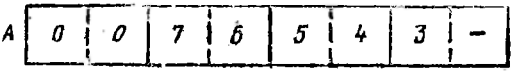
Одна команда MVO может сдвинуть число всегда на нечетное количество разрядов. Если бы мы хотели сдвинуть исходное —7 654 321 на два разряда, нужно два раза выполнить команду

```
MVO     A(4), A(3)
MVO     A(4), A(3)
```

Тогда на поле последовательно образуются сначала



а затем



§ 2.3. Фрагменты программ

Рассмотрим теперь несколько примеров составления фрагментов программ на языке ассемблера, реализующих некоторые достаточно простые действия. Эти действия мы будем предварительно описывать операторами языка АЛГОЛ-60, отождествляя ассемблерные имена с его переменными. При этом мы будем предполагать, что все имена, фигурирующие в условии задачи, уже описаны в предыдущей части программы, а соответствующие им участки памяти при исполнении этой предыдующей части заполнятся, если это необходимо, некоторыми значениями. Решение задачи должно состоять в дебавлении к программе группы машинных команд и команд ассемблера DS и DC, осуществляющих заказанные действия.

Пример 1. Y, A и X — ассемблерные имена полей длиной 2, A и X содержат некоторые значения. Требуется написать команды, реализующие оператор присваивания

$$Y := A \times X^2 - 35 \times X + 10.$$

Значения A и X предполагаются такими, что результат Y имеет не более трех значащих цифр.

Заменим наш оператор цепочкой более простых операторов, подбирая их так, чтобы каждый можно было осуществить одной машинной командой. Воспользовавшись схемой Горнера, получим очевидную последовательность

$$Y := A;\ Y := Y \times X;\ Y := Y - 35;\ Y := Y \times X;\ Y := Y + 10.$$

Казалось бы, задача уже решена, остается только написать соответствующие команды ассемблера. Но рассмотрим оператор

$$Y := Y \times X$$

Возникает первый вопрос: поместится ли значение произведения на поле Y? Мы знаем, что старое значение Y, значение X и окончательный результат не первосходят по абсолютной величине 999. Но это отнюдь не гарантирует, что все промежуточные результаты будут лежать в тех же пределах. Не вдаваясь в исследование проблемы (которая в случаях, не столь простых, как наш пример, очень трудна), предположим исходные значения A и X такими, что все промежуточные значения имеют не более трех цифр. Но если и в этом предположении мы попытаемся заменить предыдущий оператор машинной командой

```
MP      Y(2), X(2)
```

то допустим грубую ошибку. Ограничения на операнды MP требуют, чтобы длина первого операнда была больше длины второго и чтобы в его старом значении было ведущих нулей по крайней мере в два раза больше, чем длина (в байтах) второго операнда. Мы можем удовлетворить обоим условиям, взяв для

промежуточного значения $A \times X$ (и для всех следующих) поле длиной в 4 байта. Так как такого поля в условиях задачи нет, обозначим его новым именем Y1 и допишем к программе команду ассемблера

```
Y1      DS      PL4
```

определяющую новое имя и резервирующую место в памяти. Так как аналогичные соображения относятся и ко второму умножению, переделаем последовательность действий с учетом использования Y1

$$Y1 := A;\quad Y1 := Y1 \times X;\quad Y1 := Y1 - 35;\quad Y1 := Y1 \times X;$$
$$Y := Y1;\quad Y := Y + 10.$$

Нам понадобятся также две постоянные, имеющие значения целых —35 и 10. Их можно образовать с помощью команд ассемблера DC и использовать в программе присвоенные им имена. Можно также определить их с помощью литералов, написав в позиции второго операнда машинной команды вслед за знаком равенства выражение, аналогичное тому, которое нужно было бы написать на поле операндов команды ассемблера DC. Компилятор ассемблера сам допишет в конце программы машинный код соответствующей постоянной, а в адресную часть команды процессора поместит смещение и номер базисного регистра, определяющие адрес этой постоянной. (Подробнее о литералах см. [5, с. 27].)

Теперь остается только заменить операторы машинными командами

```
ZAP     Y1, A(2)          Y1 := A
MP      Y1, X(2)          Y1 := Y1 × X
AP      Y1, =P'-35'       Y1 := Y1 - 35
MP      Y1, X(2)          Y1 := Y1 × X
ZAP     Y(2), Y1          Y := Y1
AP      Y(2), =P'10'      Y := Y + 10
```

Мы опустили всюду около имени Y1 явное указание длины, так как знаем, что на основании его описания ассемблер использует характеристику длины 4.

Пример 2. Поля Y и A имеют длину 4 байта, а поля B, C, E — 3 байта каждое. На A, B, C, E лежат десятичные упакованные числа. Нужно написать команды, реализующие оператор

$$Y := A \times B \div C - E.$$

Числа A, B, C, E предполагаются такими, что промежуточное значение $A \times B \div C$ и окончательный результат имеют не более семи значащих цифр.

Оценим прежде всего величину поля, необходимого для первого промежуточного значения A × B. Так как A и B могут иметь соответственно 7 и 5 цифр, в произведении их может быть 12. Поэтому минимальная достаточная длина равна 7. (Вообще, минимальная длина поля, необходимого для размещения любого произведения сомножителей, имеющих длины K и M, есть M + K). Пусть Y1 — такое поле. Оно достаточно также для размещения частного и остатка от деления A × B на C. Действительно, длина C (а следовательно, и остатка) есть 3, а по предположению частное разместится на оставшихся четырех байтах. Считая, что характеристики длин имен Y, Y1, A, B, C, E равны длинам соответствующих полей, напишем

```
ZAP     Y1, A        Y1 := A
MP      Y1, B        Y1 := A × B
DP      Y1, C        Y1 := Y1 ÷ C
SP      Y1(4), E     Y1 := Y1 — E
ZAP     Y, Y1(4)     Y := Y1
```

Нужно добавить к программе также команду

```
Y1      DS      PL7
```

резервирующую рабочее поле длиной в 7 байтов.

Пример 3. На полях A и B одинаковой длины 6 находятся упакованные десятичные числа. Нужно на поле C той же длины образовать упакованное число по правилу

$$C := \mathrm{abs}(A) \times \mathrm{sign}(B).$$

Другими словами, на поле C нужно объединить цифровую часть A (т. е. все, кроме последнего полубайта) со знаком числа B (последний полубайт). Для этого можно воспользоваться командой MVO*. Но так как последняя может отделить от изображения числа его часть лишь по границе байта, предварительно следует сдвинуть цифры A на один разряд влево. Предположим, что для этого мы можем воспользоваться двумя байтами, расположенными непосредственно перед байтом с адресом C

```
        ZAP     C — 2(8), A          запись A на расширенное влево C
*
        MP      C — 2(8), =P'10'     сдвиг A на один разряд влево
*
        ZAP     C + 5(1), B + 5(1)   запись в последний байт поля C младшей цифры
*                                    и знака числа B
*
        MVO     C, C — 1             сдвиг C на один разряд вправо
*
```

* Используя одну из логических операций, нашу задачу можно решить двумя командами.

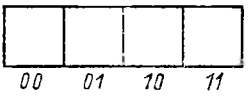
Хотя нам и достаточно для размещения сдвинутого влево на один разряд числа одного дополнительного байта, команда MP, имея второй операнд длины два, требует в первом сомножителе наличия четырех ведущих нулей. Поэтому поле C необходимо удлинить влево по крайней мере на два байта.

Заметим также, что комментарий во входной программе на языке ассемблера можно размещать не только на поле комментария, но и занимать для него отдельные строки, в первых позициях которых написан символ звездочка. Примером могут служить пять строк в тексте нашего примера. В такой строке комментарием считается все, что расположено в позициях от 2 до 71 включительно.

Для рассмотрения более интересных примеров нам необходимо познакомиться с командами переходов. На первых порах мы ограничимся командой BC (Branch on Condition) перехода по коду условия в PSW, которая в явном формате ассемблера имеет вид

BC M, $D_2(X_2, B_2)$

Число M, которое может принимать значения от 0 до 15 в десятичном выражении, означает не номер общего регистра, а четырехбитовый код, называемый маской перехода. Четыре возможных значения CC в PSW отображаются на четыре разряда маски по следующей схеме:



Мы будем говорить, что маска соответствует некоторому значению CC, если ее соответствующий разряд содержит 1. Задают маску обычно самоопределенным термом. Например, каждое из следующих выражений

B'1100' X'C' 12

определяет одну и ту же маску соответственно в двоичном, 16-ричном и десятичном виде. На поле второго операнда указывается адрес возможного перехода. Он задается либо тремя абсолютными выражениями D_2, X_2, B_2, либо в неявном формате в виде $S_2(X_2)$ или, при отсутствии индекса, в виде S_2, где S_2 означает переместимое выражение. Команда BC не производит никаких изменений ни в памяти ни в общих регистрах процессора и не меняет значения CC. Она только определяет команду-преемницу, которая будет исполняться в следующем такте. Если маска M соответствует текущему значению CC, в качестве преемницы избирается команда, расположенная по адресу, указанному в поле второго операнда, т. е. происходит переход. В противном случае сохраняется естественный порядок

и преемницей служит команда, расположенная непосредственно после команды BC.

Маски 0 и 15 соответствуют крайним случаям. M = 0 обеспечивает в любом случае выполнение следующей команды, а M = 15 — переход при любом значении CC.

В переместимом выражении S_2 можно употреблять звездочку для обозначения счетчика адреса, имеющего значение адреса первого байта команды, которая на него ссылается. Например, запись

```
A       AP      X, =P'1'
        BC      4, A
```

имеет тот же смысл, что

```
        AP      X, =P'1'
        BC      4, *-6
```

Адрес перехода всегда должен указывать на первый байт команды-преемницы.

Вместо команды BC и маски перехода в языке ассемблера можно пользоваться псевдокомандами перехода, т. е. названиями операций, включающими в себя маску перехода. В табл. 5

Таблица 5

Псевдокоманда	Маска перехода	Псевдокоманда	Маска перехода	Псевдокоманда	Маска перехода
BE	1000	BM	0100	BNH	1101
BZ	1000	BNL	1011	BNP	1101
BNE	0111	BNM	1011	BO	0001
BNZ	0111	BH	0010	BNO	1110
BL	0100	BP	0010	B	1111

указаны названия псевдокоманд вместе с масками подразумеваемого перехода.

Последние буквы названия имеют следующий смысл:

Equal, Zero, Low, Minus, High, Plus, Overflow.

Например, вместо

```
        BC      15, NAME
```

можно употребить более короткую форму

```
        B       NAME
```

Теперь мы получили возможность переводить на язык ассемблера условные операторы АЛГОЛа с отношением в позиции логического выражения. Рассмотрим оператор вида

if ⟨отношение⟩ **then** ⟨оператор1⟩ **else** ⟨оператор2⟩

Эквивалентная ему программа на ассемблере может быть написана по следующей схеме:

```
⟨программа, вырабатывающая значение логического выражения
в регистре СС слова состояния процессора⟩
    BC      ⟨нет⟩, OP2
⟨программа оператора 1⟩
    B       NEXT
OP2: ⟨программа оператора 2⟩
NEXT: ⟨программа следующих операторов⟩
```

Поясним два первые элемента схемы. Мы знаем, что некоторые машинные команды вырабатывают в регистре СС процессора код условия, характеризующий результат операции. Этим можно воспользоваться для вычисления значений логических выражений. Разбив множество всех возможных значений СС на две части, будем представлять значение «да» любым кодом одного подмножества, а «нет» — любым кодом другого. Разбиение существенно зависит от используемых машинных команд.

Например, для вычисления отношения

$$A \neq B,$$

где A и B — упакованные десятичные числа, проще всего воспользоваться командой сравнения

```
    CP      A, B
```

Тогда значению «да» будет отвечать любой из кодов {01, 10}, а значению «нет» — код 00. Код 11 командой сравнения не вырабатывается и с равным успехом может быть отнесен как к случаю «да», так и к случаю «нет». Но то же отношение можно вычислять командой

```
    SP      A, B
```

Здесь значению «да» будет соответствовать один из кодов {01, 10, 11}, а значению «нет» — код 00.

Сказанное применимо к логическому выражению любой структуры (см. по этому поводу гл. 4).

Второй элемент схемы означает команду перехода к началу программы второго оператора по маске, соответствующей любому коду из множества «нет». Например, если решено вычислять отношение $A \neq B$ первым способом, то двоичное представление такой маски есть

$$B'1001' \quad \text{или} \quad B'1000'.$$

Остальные элементы схемы в пояснениях не нуждаются. Операторы, составляющие альтернативы, в свою очередь, могут быть условными и программироваться по той же самой схеме. Нужно

также иметь в виду, что точное следование приведенной схеме не всегда дает самую экономную программу. В частности, это происходит, если один из операторов — пустой или оператор перехода. Например оператор,

if X > Y **then** Z := T

следует программировать так:

```
CP      X, Y      B'10' — «да»
BNH     *+10      переход по «нет»
ZAP     Z, T
```

а оператор

if X ⩾ Y **then go to** NAME

так:

```
        CP      X, Y      B'00' или B'10' — «да»
        BNL     NAME      переход по «да»
```

Пример 4. Поле X содержит десятичное упакованное значение. Нужно на поле Y записать значение, определяемое следующим условным оператором:

if X < 0 **then** Y := — 1 **else if** X > 0 **then** Y := 1 **else** Y := 0.

Если слепо следовать предложенной схеме, получим программу из девяти команд. Если мы воспользуемся тем, что команда

```
        CP      X, =P'0'
```

вырабатывает один из трех возможных кодов, мы получим программу из восьми команд

```
        CP      X, =P'0'
        BH      H
        BE      E
        ZAP     Y, =P'-1'     X < 0
        B       OUT
H       ZAP     Y, =P'1'      X > 0
        B       OUT
E       ZAP     Y, =P'0'      X = 0
OUT     ...
```

Если же мы сообразим, что в нашем случае можно совместить присваивание значения Y с выработкой кода СС, то сведем дело к шести командам

```
        ZAP     Y, X          B'00' при X = 0, B'01' при
                              X < 0, B'10' при X > 0
        BE      OUT           при X = 0
        BH      H             при X > 0
```

```
        ZAP     Y, =P'-1'       при X<0
        B       OUT
H       ZAP     Y, =P'1'        при X>0
OUT             ...
```

Последний способ может привести к прерыванию программы, если значение X не помещается на поле Y*.

Пример 5. На полях N и M длиной в 2 байта каждое находятся десятичные упакованные положительные числа n и m, причем $m > n \geqslant 2$. Нужно на поле R длиной 16 вычислить биномиальный коэффициент C_m^n. По известной формуле

$$C_m^n = \frac{m \times (m-1) \times \ldots \times (m-n+1)}{1 \times 2 \times \ldots \times n}.$$

Из этого следует

$$C_m^k = C_m^{k-1} \times \frac{m-k+1}{k}; \qquad C_m^1 = m.$$

Обозначив $m-k+1$ через P, можем свести вычисление C на поле R к последовательным умножениям и делениям

```
R:=M;  P:=M;  K:=1;
LOOP: P:=P-1;  K:=K+1;  R:=R×P;  R:=R÷K;
                  if K<N then go to LOOP.
```

При этом деление на K будет всегда происходить без остатка. Для переменных P и K необходимы поля по 2 байта. Зарезервируем их командами ассемблера

```
P       DS      PL2
K       DS      PL2
```

Предполагая, что R, M и N имеют соответствующие характеристики длин, перейдем к записи на языке ассемблера

```
        ZAP     R, M            R:=M
        ZAP     P, M            P:=M
        ZAP     K, =P'1'        K:=1
LOOP    SP      P, =P'1'        P:=P-1
        AP      K, =P'1'        K:=K+1
        MP      R, P            R:=R×P
        DP      R, K            R:=R÷K, остаток 2 байта
        ZAP     R, R(14)        распространение частного
        CP      K, N            IF K<N THEN
        BL      LOOP            TO LOOP
```

* Если перед исполнением программы будет замаскировано прерывание по десятичному переполнению, результат будет верен в любом случае.

Обращаем внимание на то, что хотя остаток от деления равен нулю, для него на поле R будет отводиться два байта, а частное всегда будет занимать 14 левых байтов. Результат поместится на поле R далеко не при всяких возможных значениях M и N: максимальный коэффициент при M = 999 и N = = 500 имеет порядок 10^{300}. Если M и N имеют недопустимые значения, при выполнении программы произойдет либо прерывание C при выполнении команды MR, либо прерывание ДД при выполнении команды DP.

§ 2.4. Простые программы

Мы рассмотрели ряд примеров, иллюстрирующих фрагменты входных программ. Познакомимся с требованиями, которым должна удовлетворять законченная входная программа, предъявляемая компилятору ассемблера. Поскольку входная программа в нормальном случае включает в себя команды чтения исходных данных и печати результатов, ее структура зависит от типа операционной системы, используемой на конкретной машине. Наиболее распространенными являются системы ДОС и ОС. Мы в дальнейшем будем считать, что наш ассемблер функционирует в рамках ОС и будем пользоваться макрокомандами супервизора именно этой системы. Макрокоманда по форме записи похожа на машинные команды или команды ассемблера. При компиляции она заменяется группой машинных команд — макрорасширением, которая обычно содержит команду обращения к процедурам супервизора, выполняющим привилегированные функции. Замена происходит на основании описаний макрокоманд, хранящихся в специальной библиотеке. Библиотека допускает изменения, что дает нам право пользоваться кроме общепринятых (системных) макрокоманд дополнительными, удобными в том или ином конкретном случае. В Вычислительном центре Ленинградского университета применяется ряд таких макрокоманд, упрощающих составление входной программы. Определения этих макрокоманд приведены в приложении 3.

Обычно машинная программа имеет структуру, показанную на рис. 4. Предположим, что она занимает отрезок памяти не более чем в 4096 байтов, так что любой ее элемент можно адресовать, варьируя нужным образом смещение, с помощью одного базисного регистра, содержащего адрес первого байта программы. Такой регистр мы в дальнейшем будем называть базисным регистром программы. Этот регистр назначается автором программы из числа общих регистров, отличных от регистров с номерами 0, 1, 13, 14, 15, которые используются макрокомандами ОС специальным образом. Выполнение машинной программы, загруженной в память операционной системой, начинается с первой команды группы, отмеченной на схеме

словом ВХОД *. Эта группа должна запомнить данные, сообщаемые управляющей программой операционной системы в регистрах 13 и 14, загрузить адрес в базисный регистр и выделить область сохранения для программы (см. гл. 4). Все названные действия обеспечиваются макрокомандой

```
имя     PROC     12
```

открывающей входную программу. Число 12 на поле операндов означает, что в качестве базисного регистра программы нами избран регистр 12.

Группа команд ВЫХОД должна исполняться после того, как работа программы закончится. Она информирует операционную систему о том, что выполнение программы завершилось нормальным образом. Во входной программе для этой цели можно воспользоваться макрокомандой

```
        EXIT
```

Конец входной программы необходимо отметить командой ассемблера

```
        END
```

Если в нашей программе предусмотрено чтение входной информации с перфокарт, можно употреблять две следующие макрокоманды:

```
        OPENIN  имя блока управления, адрес выхода
```

и

```
        GET  имя блока управления, адрес поля в 80 байтов.
```

Первая макрокоманда строит блок управления чтением перфокарт. Ее первый операнд — назначаемое автором программы ассемблерное имя, указывающее на начало этого блока. Второй, необязательный, операнд задает адрес той команды нашей программы, к которой следует перейти, если входная информация исчерпана. Макрокоманда OPENIN должна выполняться только один раз и обязательно до первого исполнения макрокоманды GET. Последняя же может исполняться многократно, перенося каждый раз содержимое очередной перфокарты из читающего устройства машины на 80-байтное поле, адрес начала которого указан во втором операнде GET. При этом каждый из 80 символов, записанных во внешнем коде в колонках перфокарты, переводится во внутренний код. Например, если в первых трех колонках первой карты отперфорированы три цифры 5, а остальные не содержат перфораций, то после первого ис-

* В общем случае группа ВХОД может располагаться не в начале программы.

полнения GET на поле, указанном вторым параметром, появится код

11110101	11110101	11110101	01000000	01000000	. . .

т. е. в трех первых байтах — цифра 5 в зонном формате, а в остальных 77 позициях — внутренний код пробела.

Для того чтобы напечатать результаты программы, можно воспользоваться макрокомандами

OPENOUT имя блока управления печатью

и

PUT имя блока управления печатью, начало строки.

Макрокоманда OPENOUT строит блок управления печатью и должна выполняться один раз. Макрокоманда PUT печатает на алфавитно-цифровом печатающем устройстве в виде строки длиной в 128 позиций информацию, хранящуюся во внутреннем коде в последовательности из 128 байтов, указанной вторым параметром макрокоманды PUT. Точнее, второй параметр — адрес байта, предшествующего первой печатаемой литере и содержащего управляющий символ. Этот символ задает величину продвижения бумаги после печати строки (см. [7, гл. 16]).

С помощью команды PUT можно печатать не только цифровую информацию, но и любые тексты, состоящие из основных и дополнительных символов ассемблера. Для подготовки таких текстов можно пользоваться командой ассемблера DC, поле операндов которой начинается с буквы C (Character). Например, команда

```
A       DC      CL20'ТЕКСТ'
```

заготовит последовательность из 20 байтов, первые пять из которых будут содержать внутренние коды букв ТЕКСТ, а остальные пятнадцать — внутренние коды пробелов.

Для иллюстрации всего сказанного приведем пример законченной входной программы, которая читает последовательность трехзначных целых чисел без знака, находит максимальное в последовательности число и печатает на выходном печатающем устройстве строку

МАКСИМУМ = xxx

где знаки xxx замещают три цифры искомого максимума:

```
         PRINT    NOGEN      выключение пе-
*                            чати расшире-
*                            ний
MAXIMUM  PROC     12         вход в про-
*                            грамму
         OPENIN   IN, END    образование
```

```
*                                                    блока управле-
*                                                    ния вводом
          OPENOUT   OUT                              образование
*                                                    блока управле-
*                                                    ния выводом
          ZAP       MAX, =P'0'                       MAX:=0
LOOP      GET       IN, F80                          чтение карты
*                                                    с очередным
*                                                    числом
          PACK      NUM, F80(3)                      перевод числа
*                                                    в упакованный
*                                                    формат
          CP        MAX, NUM                         проверка на
*                                                    максимум
          BNL       LOOP
          ZAP       MAX, NUM                         замена значе-
*                                                    ния MAX
          B         LOOP
END       UNPK      C+30(3), MAX                     перевод MAX
*                                                    в зонный фор-
*                                                    мат
          OI        C+32, X'F0'                      замена кода
*                                                    знака плюс ко-
*                                                    дом зоны
          PUT       OUT, C                           печать резуль-
*                                                    тата
          EXIT                                       выход из про-
*                                                    граммы
F80       DS        80C                              поле для оче-
*                                                    редной перфо-
*                                                    карты
MAX       DS        PL2
NUM       DS        PL2
C         DC        CL21' '                          21 пробел
          DC        CL108'МАКСИМУМ='
          END
```

Первую строку программы составляет команда ассемблера PRINT, управляющая печатью листинга программы. Операнд NOGEN означает, что мы просим не включать в состав листинга расширений макрокоманд, чтобы сделать листинг более коротким и наглядным.

Макрокоманда PROC образует вход в программу и назначает общий регистр 12 базисным регистром программы MAXIMUM.

Макрокоманда OPENIN заводит таблицу управления вводом, помечает ее первый байт ассемблерным именем IN и вносит в эту таблицу адрес END команды программы, к которой

нужно перейти по исчерпании перфокарт с входными данными.

Макрокоманда OPENOUT заводит таблицу управления выводом и помечает ее первый байт ассемблерным именем OUT.

Машинная команда ZAP заносит 0 на поле MAX.

Следующие далее макрокоманда GET и пять машинных команд составляют циклическую часть программы MAXIMUM, повторяющуюся столько раз, сколько карт с числами будет подложено к программе. При этом предполагается, что каждое число имеет ровно три цифры, отперфорированные в трех первых колонках карты. Когда после очередного перехода к макрокоманде GET последняя обнаружит, что карты с данными окончились, она передаст управление по адресу END, внесенному ранее в таблицу управления вводом. Команда UNPK переведет значение MAX в зонный формат и запишет его в три байта, расположенные на поле C сразу после байта с внутренним кодом знака равенства. В левой половине младшего байта результата, имеющего адрес C + 32, появится код знака плюс B'1100', выработанный командой AP. Команда OI (см. § 4.1) заменяет его па B'1111'. Поэтому макрокоманда PUT напечатает правильно все три цифры результата. При этом первая буква М слова МАКСИМУМ будет расположена на бумаге на 20 позиций правее левого края поля печати.

Для того чтобы компилировать и исполнить нашу программу, нужно перенести ее на перфокарты и добавить к ней еще две группы карт: карты с операторами языка управления заданиями, обеспечивающими обращение к компилятору ассемблера, и карты с входными данными для машинной программы, а затем передать всю пачку оператору на машине для исполнения в пакетном режиме.

Расположение карт в колоде должно быть следующим:

```
//                JOB       ...                          1
//                EXEC      ASMFCLG                      2
                  карты с входной программой
//GO.IN           DD        *                            3
                  карты с входными данными для программы MAXIMUM
//GO.OUT          DD        SYSOUT=A                     4
//GO.SYSUDUMP     DD        SYSOUT=A                     5
//                                                       6
```

Назначение управляющих карт следующее. Карта № 1 составляет начало описания задания системе в пакетном режиме. Она содержит ряд параметров, зависящих от конфигурации конкретной операционной системы. Карта № 2 вызы-

вает стандартную процедуру операционной системы, включающую компиляцию, редактирование и исполнение программы, написанной на языке ассемблера. Карта № 3 информирует систему о том, что вслед за ней располагаются входные данные компилируемой программы. Карта № 4 предписывает поместить информацию, выдаваемую макрокомандой PUT (в нашем случае это одна строка) на системное устройство печати. Карта № 5 необходима лишь в том случае, когда при исполнении машинной программы процессор обнаружит ошибку и программист желает при этом получить подробную информацию о ситуации, включающую 16-ричное изображение PSW и кодов, находящихся в занятых программой ячейках. Карта № 6 является признаком конца описания задания.

Подробную информацию о правилах компиляции, редактирования и исполнения программы можно найти в [9] (см. также гл. 5 настоящего руководства).

Практически каждая написанная человеком программа содержит ошибки и нуждается в отладке. Синтаксические ошибки, допущенные во входной программе, обнаруживаются компилятором и указываются в листинге, так что устранение их не составляет особого труда. Гораздо труднее найти семантические ошибки, обнаруживающие себя лишь при исполнении скомпилированной программы. Даже если такая ошибка приведет к прерыванию программы и место прерывания будет указано в информации, выдаваемой системой при наличии управляющей карты дампа (карта № 5 в нашем примере), это далеко не всегда позволяет просто обнаружить ошибочную команду или постоянную. Простейшим и достаточно эффективным методом локализации семантических ошибок служат дополнительные макрокоманды отладочной печати, расставляемые автором в определенных точках программы. Рассматривая результаты, выданные этими макрокомандами при тестовых входных данных, и сравнивая их с правильными результатами, найденными каким-либо другим путем, он может значительно сузить область, в которой следует искать ошибку. Обычно для названных целей используется системная макрокоманда SNAP, которая печатает состояние ячеек памяти и регистров процессора в виде последовательностей 16-ричных цифр, изображающих каждая код в четыре бита. Эта макрокоманда имеет много операндов. Исчерпывающую информацию о ней можно найти в [10]. В простейшем случае макрокоманда имеет вид

$$\text{SNAP} \quad \text{DCB}=\text{адрес блока управления},$$
$$\text{STORAGE}=(\text{начальный адрес}, \text{конечный адрес})$$

Каждое исполнение такой макрокоманды приводит к печати участка памяти, указанного в операнде STORAGE. Для ука-

зания начала и конца участка обычно используются перемещаемые выражения. В операнде DCB указывается адрес блока управления выводом, который открывается в начале программы макрокомандой OPENSNAP, аналогичной макрокоманде OPENOUT. Именно, если в начале нашей программы расположена макрокоманда

```
OPENSNAP      OUT1
```

то операнд DCB во всех экземплярах макрокоманды SNAP должен быть таким:

```
DCB = OUT1
```

Кроме того, в конце описания задания (т. е. перед картой № 6 в нашем примере) нужно поместить еще одну управляющую карту

```
GO.OUT1      DD                SYSOUT = A
```

Глава 3

ОПЕРАЦИИ НАД ДВОИЧНЫМИ ЧИСЛАМИ, ЦИКЛЫ И ПРЯМОУГОЛЬНЫЕ МАССИВЫ

§ 3.1. Представление двоичных целых чисел и операции над ними

Наряду с десятичными числами в машинах серии ЕС ЭВМ предусмотрено представление целых чисел в двоичной системе. Преимущества двоичной формы — компактность и бо́льшая скорость машинных операций. Для изображения двоичных чисел используется дополнительный код с количеством битов n, равным 16, 32 и иногда 64. Мы познакомимся с особенностями этого кода при $n = 4$. Разряды кода будем нумеровать слева направо цифрами от 0 до 3 (а в общем случае до $n - 1$). Разряд 0 хранит знак числа. Цифра 0 в этом разряде означает, что число положительно, а цифра 1 — что оно отрицательно. Дополнительный код положительного числа совпадает с его записью в двоичной системе. При $n = 4$ дополнительный код числа 7 — максимального из представимых в нашем случае — есть 0111. В общем же случае это максимальное число есть $2^{n-1} - 1$.

Чтобы получить дополнительный код отрицательного числа, нужно в коде его абсолютного значения поменять все разряды на обратные, а затем добавить 1 к его младшему разряду.

Например, дополнительный код числа 6 есть 0110. Дополнительный код числа —6 можно получить так:

$$\begin{array}{r} 1001 \\ +\ 1 \\ \hline 1010 \end{array}$$

Это же правило применимо для перемены знака с минуса на плюс

$$\begin{array}{r} 0101 \\ +\ 1 \\ \hline 0110 \end{array}$$

Минимальным отрицательным числом, представимым дополнительным кодом при $n = 4$, является число —8, имеющее код

1000. В общем случае такое число есть -2^{n-1}. К этому числу (и только к нему из всех представимых чисел) правило перемены знака неприменимо.

Дополнительные коды обладают свойством, упрощающим реализацию в арифметическом устройстве процессора сложение чисел разных знаков: если мы сложим дополнительные коды чисел x и y как n-разрядные двоичные положительные числа по модулю 2^n (т. е. отбросив единицу переноса из нулевого разряда, если она появится), мы получим дополнительный код алгебраической суммы x + y (конечно, лишь в том случае, если x + y есть представимое число). Например

−8	+	1000	+7	+	0111
+5		0101	−5		1011
−3		1101	2		0010

Если же сумма выходит за пределы допустимого диапазона (от -2^{n-1} до $2^{n-1}-1$), результат неверен. Такие случаи можно распознать, следя за переносами из разряда один в разряд ноль и из разряда ноль. В нормальном случае оба переноса или есть, или отсутствуют. В случае переполнения имеет место только один из переносов. Например

+5	+	0101	−4	+	1100
+4		0100	−5		1011
+9		1001	−9		0111
		←			←

В случае переполнения правильный результат можно восстановить, дописав к сумме кодов слева еще один разряд со значением, противоположным нулевому разряду.

Заметим, что число нуль в дополнительном коде имеет только одно представление. Если мы попытаемся по общему правилу изменить его знак, то получим прежний код

1111
+1
0000

Машинные операции над двоичными целыми числами имеют два типа операндов: длинные (n = 32) и короткие (n = 16). Длинные операнды хранятся в машинных словах (т. е. в полях из четырех байтов, начинающихся с байта с адресом кратным четырем) или в общих регистрах. Короткие операнды хранятся в машинных полусловах (т. е. в полях из двух байтов, первый из которых имеет четный адрес). Максимальное значение длинного числа есть $2^{31}-1 = 2\,147\,483\,647$, а короткого числа — $2^{15}-1 = 32\,767$. Кроме того, в операции деления для представления делимого и в операции умножения длинных чисел для представления произведения используется

Таблица 6

Название операции	Операнды	Прерывания	Код условия				Описание операции	Время
			00	01	10	11		
1	2	3	4	5	6	7	8	9
LR	R_1, R_2						$R_1 := R_2$	1,02
L	R_1, D_2 (X_2, B_2)	A, C					$R_1 := D_2$ (X_2, B_2)	4,47
LH	R_1, D_2 (X_2, B_2)	A, C					$R_1 := D_2$ (X_2, B_2)	5,48
LTR	R_1, R_2		$R_1 = 0$	$R_1 < 0$	$R_1 > 0$		$R_1 := R_2$	1,02
LCR	R_1, R_2	П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := -R_2$	1,02
LPR	R_1, R_2	П	$R_1 = 0$		$R_1 > 0$	Пер	$R_1 :=$ abs (R_2)	1,02
LNR	R_1, R_2		$R_1 = 0$	$R_1 < 0$			$R_1 := -$ abs (R_2)	1,02
LM	R_1, R_3, D_2 (B_2)	A, C					$R_1 := D_2$ (B_2); $R_1 + 1 := D_2 + 4$ (B_2); ...	7,71 *
ST	R_1, D_2 (X_2, B_2)	A, C, З					D_2 (X_2, B_2) := R_1	4,87
STH	R_1, D_2 (X_2, B_2)	A, C, З					D_2 (X_2, B_2) := $[R_1]_{16}$ **	4,88
STM	R_1, R_3, D_2 (B_2)	A, C, З					D_2 (B_2) := R_1; $D_2 + 4$(B_2) := := ($R_1 + 1$); ...	8,12 *
AR	R_1, R_2	П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 + R_2$	1,01
A	R_1, D_2 (X_2, B_2)	A, C, П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 + D_2$ (X_2, B_2)	4,46
AH	R_1, D_2 (X_2, B_2)	A, C	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 + D_2$ (X_2, B_2)	5.28
ALR	R_1, R_2		$0 \wedge \neg B$	$\neg 0 \wedge \neg B$	$0 \wedge B$	$\neg 0 \wedge B$	$R_1 := R_1 + R_2$ ***	1,01
AL	R_1, D_2 (X_2, B_2)	A, C	$0 \wedge \neg B$	$\neg 0 \wedge \neg B$	$0 \wedge B$	$\neg 0 \wedge B$	$R_1 := R_1 + D_2$ (X_2, B_2) ***	4,46
SR	R_1, R_2	П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 - R_2$	1,01
S	R_1, D_2 (X_2, B_2)	A, C, П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 - D_2$ (X_2, B_2)	4,47
SH	R_1, D_2 (X_2, B_2)	A, C, П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	$R_1 := R_1 - D_2$ (X_1, B_2)	5,69
SLR	R_1, R_2		$0 \wedge \neg B$	$\neg 0 \wedge \neg B$	$0 \wedge B$	$\neg 0 \wedge B$	$R_1 := R_1 - R_2$ ***	1,01
SL	R_1, D_2 (X_2, B_2)	A, C	$0 \wedge \neg B$	$\neg 0 \wedge \neg B$	$0 \wedge B$	$\neg 0 \wedge B$	$R_1 := R_1 - D_2$ (X_2, B_2) ***	4,47

MR	R_1, R_2	C					$\{R_1, R_1+1\} := (R_1+1) \times R_2$	17,36
M	R_1, $D_2(X_2, B_2)$	A, C					$\{R_1, R_1+1\} := (R_1+1) \times D_2(X_2, B_2)$	18,27
MH	R_1, $D_2(X_2, B_2)$	A, C					$R_1 := [R_1 \times D_2(X_2, B_2)]_{32}$	15,64
DR	R_1, R_2	C, Дл					$R_1+1 := \{R_1, R_1+1\} \div R_2$; R_1 := остаток	31,18
D	R_1, $D_2(X_2, B_2)$	A, C, Дл					$R_1+1 := \{R_1, R_1+1\} \div D_2(X_2, B_2)$; R_1 := остаток	32,89
CR	R_1, R_2		$R_1 = R_2$	$R_1 < R_2$	$R_1 > R_2$		Сравнение	1,02
C	R_1, $D_2(X_2, B_2)$	A, C	$R_1 = D_2(X_2, B_2)$	$R_1 < D_2(X_2, B_2)$	$R_1 > D_2(X_2, B_2)$		Сравнение	4,46
CH	R_1, $D_2(X_2, B_2)$	A, C	$R_1 = D_2(X_2, B_2)$	$R_1 < D_2(X_2, B_2)$	$R_1 > D_2(X_2, B_2)$		Сравнение	5,28
SLA	R_1, $D_2(B_2)$	П	$R_1 = 0$	$R_1 < 0$	$R_1 > 0$	Пер	Арифметический сдвиг влево	3,66 ****
SRA	R_1, $D_2(B_2)$		$R_1 = 0$	$R_1 < 0$	$R_1 > 0$		Арифметический сдвиг вправо	3,65 ****
SLDA	R_1, $D_2(B_2)$	С, П	$\{R_1, R_1+1\} = 0$	$\{R_1, R_1+1\} < 0$	$\{R_1, R_1+1\} > 0$	Пер	Двойной сдвиг влево	4,87 ****
SRDA	R_1, $D_2(B_2)$	C	$\{R_1, R_1+1\} = 0$	$\{R_1, R_1+1\} < 0$	$\{R_1, R_1+1\} > 0$		Двойной сдвиг вправо	4,47 ****
CVB	R_1, $D_2(X_2, B_2)$	A, C, Дл, Д					$R_1 := [D_2(X_2, B_2)]$	28,02
CVD	R_1, $D_2(X_2, B_2)$	A, C, З					$D_2(X_2, B_2) := [R_1]$	16,65
LA	R_1, $D_2(X_2, B_2)$						$R_1 := D_2 + (X_2) + (B_2)$	3,25

* Время указано для случая трех пар операндов.
** Результат составляют 16 младших битов общего регистра.
*** Буква В обозначает выход единицы переноса из нулевого разряда.
**** Время указано для сдвига на три бита.

дополнительный код с n = 64, располагающийся в двух общих регистрах с четным и следующим нечетным номерами.

Познакомимся с таблицей машинных операций над двоичными целыми числами (табл. 6). Исчерпывающее описание каждой команды читатель может найти в [1, с. 54—83]. Здесь же мы поясним только самое существенное. Команды имеют формат RR, RX или RS и объединяются в девять групп: загрузка (Load), выгрузка (Store), сложение (Add), вычитание (Subtract), умножение (Multiply), деление (Divide), сравнение (Compare), сдвиг (Shift) и перевод (Convert).

1-я графа таблицы содержит мнемоническое название команды на языке ассемблера. Как правило, первые одна-две буквы составляют сокращенное название соответствующей операции, а буква R есть признак формата RR. Во 2-й графе приведены обозначения операндов в явном формате ассемблера. R_1 есть номер общего регистра, в котором образуется результат операции (исключение составляют команды выгрузки). D_2, X_2 и B_2 означают соответственно смещение, номер индексного и номер базисного регистра команды. В позициях R_1, R_2, D_2, X_2, B_2 можно писать абсолютные выражения (см. [5, с. 29]). В простейшем случае это — десятичные целые без знака.

3-я графа содержит список причин, могущих вызвать прерывание программы во время исполнения операции: А — адрес операнда превышает максимальный, С — нарушение спецификации операндов (нечетное R_1, неправильное выравнивание второго операнда по границам полуслов, слов или двойных слов), З — нарушение защиты памяти при записи результата, Дл — невозможность деления, П — переполнение (результат операции выходит за допустимый диапазон).

Графы 4—7 содержат правило выработки кода условия по результату команды. Команды, в которых все четыре графы пусты, не меняют код условия, выработанный другими командами.

8-я графа содержит алголоподобное описание операции. Выражение $\{R_1, R_1 + 1\}$ означает, что операнд занимает пару общих регистров: старшая часть располагается в регистре с четным номером R_1, а младшая — в регистре $R_1 + 1$ со следующим нечетным номером.

В 9-й графе приведено время в микросекундах исполнения соответствующей команды программы процессором модели 1033. Для операций сдвига и кратной загрузки или выгрузки это время зависит соответственно от величины сдвига или количества операндов.

В дальнейших примерах мы будем пользоваться обычно для команд форматов RX и RS неявным форматом второго операнда, имеющим вид

$$S(X_2) \text{ или } S,$$

где S заменяет переместимое выражение (в простейшем случае ассемблерное имя плюс или минус некоторое число), а X_2 имеет тот же смысл, что и в явном формате. Ассемблер заменяет выражение S номером базисного регистра и смещением, подставляя вместо X_2 при его отсутствии нулевой регистр.

Мы будем пользоваться также командами ассемблера DC и DS для заготовки постоянных и резервирования мест в памяти. Для действий с двоичными целыми числами удобно пользоваться командой DS следующей формы:

[имя] DS [кратность] $\left\{\begin{matrix}\text{H}\\\text{F}\\\text{D}\end{matrix}\right\}$

Здесь и в дальнейшем скобки [ ] заключают необязательный элемент записи, а скобки { } содержат написанные друг под другом альтернативные элементы.

Кратность обычно задается десятичным числом без знака и определяет количество резервируемых полей. Кратность 0 означает требование продвинуть счетчик адреса ассемблера вперед до ближайшего байта с адресом требуемой операндом четности. Отсутствие кратности эквивалентно написанию на ее месте единицы. Буквы H, F и D указывают размер и выравнивание резервируемой единицы: H — полуслово, F — слово, D — двойное слово. Если очередной свободный байт не удовлетворяет условиям выравнивания, ассемблер пропускает один или более байтов. Например, разность адресов, соответствующих ассемблерным именам B и A в записи

```
A    DS    H
B    DS    F
```

может оказаться равной либо 2 либо 4 в зависимости от того, равен 2 или 0 остаток от деления на четыре адреса, соответствующего имени A.

Заказать двоичные постоянные можно командой

[имя] DC [кратность] $\left\{\begin{matrix}\text{H}\\\text{F}\end{matrix}\right\}$ [модификаторы] 'список чисел'

Список чисел может состоять из одного или нескольких элементов, разделенных запятыми. Числа пишутся в десятичной системе со знаком или без знака. Например, команда

```
P    DC    F'-5, 15'
```

образует два длинных двоичных эквивалента чисел —5 и 15, расположенных в двух соседних словах с адресами B и B + 4. Дальнейшие сведения о форме записи чисел и структуре модификаторов можно получить в [5], раздел 3.6.1.

Для заказа постоянных можно также в позиции второго операнда (за исключением команд выгрузки и переводов) писать литерал вида

$$=\left\{\begin{matrix}\text{H}\\\text{F}\end{matrix}\right\}\ [\text{модификатор}]\ \text{'число'}$$

Более точное описание операндов команд DC, DS и литералов можно найти в [5], разделы 3.6 и 3.7, или в [9], раздел 1.34.

Перейдем теперь к обзору машинных операций над двоичными числами. Команда LH — загрузка полуслова (Halfword) — отличается от загрузки слова L тем, что ее операнд в памяти есть двоичное короткое число, которое при загрузке в регистр за счет размножения разряда знака удлиняется до длинного числа. Команда выгрузки полуслова STH записывает в полуслово, указанное адресом второго операнда, 16-битовый код из правой (младшей) части общего регистра. Например, если A — имя полуслова, содержащего короткое двоичное число —1, а B — имя некоторого машинного слова, то после выполнения двух команд

```
LH      0, A
ST      0, B
```

в машинном слове B образуется длинный дополнительный код числа —1.

Команда групповой загрузки LM (Load Multiple) загружает несколько регистров кодами из последовательности соседних машинных слов, адрес первого из которых указан на поле второго операнда. Номера загружаемых регистров составляют последовательность от R_1 до R_3 включительно по модулю 16. Например, команда

```
LM      12, 14, B
```

загружает регистры с номерами 12, 13, 14 кодами из машинных слов B, B + 4, B + 8, а команда

```
LM      14, 12, B
```

загружает регистры 14, 15, 0, 1, ..., 11, 12 кодами из пятнадцати машинных слов с адресами соответственно B, B + 4, B + + 8, ..., B + 56. Команда кратной выгрузки STM выгружает регистры подобным же образом.

Команда сложения полуслов AH образует в общем регистре сумму по модулю 2^{32} кода, находившегося в регистре, и удлиненного влево кода из полуслова, указанного адресом на поле второго операнда. Команды логического сложения ALR и AL вырабатывают результат в регистре R_1 точно по тем же правилам, что и команды соответственно AR и A. Отличие лишь в том, что при переполнении не происходит прерывания и по дру-

гому правилу вырабатывается признак результата в регистре CC. Например, код 11 устанавливается в том случае, когда результат в R_1 отличен от нулевого и при его образовании произошел перенос единицы из знакового разряда. Операции логического сложения ALR и AL и аналогичные операции логического вычисления SLR и SL предназначены для программного моделирования операций сложения и вычитания над двоичными числами, дополнительные коды которых занимают два или больше машинных слов.

Операции арифметического сдвига влево SLA (Left Algebraic) и арифметического сдвига вправо SRA (Right Algebraic) производят сдвиг влево или вправо значащей части дополнительного кода числа. При сдвиге влево на каждую позицию пропадает прежнее значение разряда один, а в освободившийся разряд тридцать один заносится 0. При сдвиге вправо пропадает значение из разряда тридцать один, а в разряд один поступает код знакового разряда. Разряд знака при этом не изменяется. Сдвигаемый код находится в регистре R_1, а величина сдвига указывается шестью младшими двоичными разрядами адреса второго операнда. Операции SLDA и SRDA производят двойной (Double) арифметический сдвиг соответственно влево и вправо. Сдвигаемый код состоит из 64 битов и занимает два общих регистра. Старшая часть находится в регистре с четным номером R_1, а младшая — в регистре со следующим нечетным номером.

Арифметический сдвиг вправо на n разрядов с точностью до округления эквивалентен умножению числа на 2^{-n}, а сдвиг влево соответствует умножению на 2^n. В последнем случае возможно переполнение. Признаком переполнения является «выталкивание» из разряда один бита, противоположного значению разряда ноль.

Рассмотрим пример использования команд сдвига. Пусть в двенадцати последних битах машинного слова A находится число x, изображенное дополнительным кодом с $n = 12$. Нужно превратить код x в обычный дополнительный код длинного числа ($n = 32$) и записать его в слово B. Эти действия можно осуществить следующими командами:

```
L       0, A     код x в правой части регистра 0
SRDA    0, 12    код x в начале регистра 1
SRA     1, 20    растянутый код x в регистре 1
ST      1, B
```

Указывая величину сдвига во второй и третьей командах, мы воспользовались одним из сокращений явного формата, допустимых в языке ассемблера. В полной записи вторая, например, команда имела бы вид

```
SRDA    0, 12(0)
```

Напомним, что общий регистр 0 не может быть ни индексным, ни базисным, и поэтому указание его на поле второго операнда означает отсутствие соответствующего слагаемого машинного адреса операнда.

В командах MR и M умножения длинных чисел первый сомножитель находится в общем регистре с нечетным номером $R_1 + 1$, а R_1 есть номер четного регистра, содержимое которого до начала операции безразлично*. Второй сомножитель находится либо в регистре R_2, четность которого безразлична, либо в машинном слове с адресом $D_2(X_2, B_2)$. Произведение представляется дополнительным кодом с $n = 64$ и размещается в двух регистрах: четном R_1 (старшая часть) и нечетном $R_1 + 1$ (младшая часть). Поэтому переполнения при умножении не может быть.

Команда MH умножения полуслова образует в регистре R_1 (четность которого безразлична) младшие 32 разряда дополнительного кода произведения короткого числа из полуслова с адресом $D_2(X_2, B_2)$ на длинное число, находившееся до операции в регистре R_1. Результат операции заведомо верен, если в регистре R_1 до операции находилось число из диапазона коротких чисел. В противном случае результат неверен, но прерывание по переполнению отсутствует.

В командах деления DR и D делимое должно быть предварительно представлено дополнительным кодом с $n = 64$ и помещено в регистры R_1 и $R_1 + 1$ (R_1 обязательно четное). Делителем служит длинное двоичное число соответственно в регистре R_2 или слове с адресом $D_2(X_2, B_2)$. Результат операции — остаток и частное, оба в дополнительном коде. Остаток образуется в регистре R_1, а частное — в регистре $R_1 + 1$. Знак остатка совпадает со знаком делимого. Если делимое и делитель таковы, что частное выходит за диапазон длинных чисел, деление не производится и происходит прерывание по невозможности деления.

Пусть, например, в машинных словах A и B хранятся делимое и делитель, а остаток и частное мы хотим получить соответственно в словах C и D. Для этого можно воспользоваться командами

```
        L       0, A    делимое в регистре 0
        SRDA    0, 32   удлинение делимого размножением
*                       разряда знака
        D       0, B    остаток в регистре 0, частное в ре-
*                       гистре 1
```

---

* За исключением того случая команды MR, когда в регистре с этим номером содержится второй сомножитель Например, команда MR 2,2 образует в регистрах 2 и 3 произведение чисел, до выполнения команды хранившихся в регистрах 2 и 3.

```
ST      0, C    остаток в слове C
ST      1, D    частное в слове D
```

Команды сравнения C, CH, CR не производят никаких изменений ни в общих регистрах, ни в памяти. Они только вырабатывают в регистре CC один из возможных кодов условия: 00, 01, 10 в зависимости от того, равен первый операнд второму, меньше второго или больше второго. В командах CR и C оба операнда являются длинными двоичными числами, а в команде CH первый и второй операнды имеют разную длину.

Команда CVB предназначена для перевода упакованного десятичного числа (второй операнд) в двоичное длинное число. Второй операнд должен иметь длину восемь и располагаться в двойном машинном слове. При выполнении этой операции, кроме прерываний по неправильным данным, по спецификации (адрес не кратен 8) или по превышению адреса, может произойти прерывание по невозможности деления в том случае, если двоичный эквивалент десятичного числа выходит за пределы диапазона длинных двоичных чисел.

Обратная операция CVD переводит число, изображенное дополнительным кодом в регистре R_1, в десятичную упакованную форму длины 8, записываемую в двойное машинное слово с адресом $D_2(X_2, B_2)$.

Операция LA — загрузка адреса — состоит в том, что машинный адрес, определяемый полем второго операнда, записывается в регистр R_1. В частности, эту команду удобно использовать для занесения в общий регистр положительных целых чисел, не превосходящих 4095, т. е. наибольшего числа, которое можно записать в форме смещения D_2. Например, в результате исполнения команды

```
LA      1, 395
```

в общий регистр 1 поступит дополнительный код числа 395. Эту же операцию можно употребить для образования суммы трех положительных чисел: постоянной на поле смещения и еще двух слагаемых в общих регистрах, отличных от нулевого. При этом надо помнить, что машинный адрес вычисляется по модулю 2^{24}. Например, по команде

```
LA      1, 395(2, 12)
```

в регистр 1 поступят младшие 24 двоичных разряда суммы дополнительного кода числа 395 и кодов в регистрах 2 и 12.

В заключение обзора рассмотрим два примера на действия с двоичными целыми числами.

Пример 1. Пусть в машинных словах A, B и X хранятся длинные двоичные числа. Нужно изменить значение X по правилу

$$X := \mathrm{abs}(A \times X^2 + B \times X - 25) \div 2^5.$$

При этом предполагается, что как окончательный результат, так и все промежуточные не выходят из диапазона длинных двоичных чисел.

L	1, A	$R1 := A$
M	0, X	$R0, R1 := A \times X$
A	1, B	$R1 := A \times X + B$
M	0, X	$R0, R1 := (A \times X + B) \times X$
AH	1, =H′ — 25′	$R1 := (A \times X + B) \times X - 25$
LPR	1, 1	$R1 := \mathrm{abs}(R1)$
SRA	1, 5	$R1 := R1 \div 32$
ST	1, X	$X := R1$

В этом примере и во всех нижеследующих мы на поле комментария идентификаторами R0, R1, R2, ..., обозначаем регистры 0, 1, 2, ..., рассматриваемые как переменные в смысле АЛГОЛ-60.

Пример 2. Пусть в полуслове X хранится некоторое двоичное число. Нужно в общем регистре 0 образовать число sign(X)

LH	0, X	$R0 := X$
SRDA	0, 32	результат при $X \leqslant 0$
BNH	*+8	выход при $X \leqslant 0$
LA	0, 1	$R0 := 1$ при $X > 0$

Действительно, после выполнения второй команды код числа X передвинется в регистр 1, а в регистре 0 останется его размноженный знаковый разряд. Если $X < 0$, это — дополнительный код числа —1, а при $X = 0$ — код числа 0. Поэтому результат нужно изменить лишь в том случае, когда $X > 0$. Эту операцию выполняет четвертая команда.

Предлагаем читателю подумать над вопросом: можно ли в этом примере заменить вторую команду на такую

SRA	0, 32	?

§ 3.2. Числа с плавающей точкой

При вычислениях, связанных с решением алгебраических и функциональных уравнений, приходится оперировать с рациональными числами, изменяющимися в очень широком диапазоне. Принципиально все нужные действия можно свести к операциям над целыми числами. Но в большинстве случаев такой путь неприемлем, так как требует длинных программ и приводит к большим потерям машинного времени.

Поэтому в системе ЕС ЭВМ предусмотрены представление чисел с плавающей точкой и группа машинных операций с этими числами. Существуют две формы чисел с плавающей

точкой: короткая и длинная. В дальнейшем мы будем называть первые числами типа E, а вторые — числами типа D.

Множество чисел типа E состоит из числа 0 и всех рациональных чисел x, которые можно представить произведением трех сомножителей:

$$x = \mathrm{sign}(x)\, 16^p m.$$

Здесь p обозначает целое число, удовлетворяющее неравенству

$$-64 \leqslant p \leqslant 63,$$

а m есть любое рациональное из замкнутого промежутка

$$16^{-6} \leqslant m \leqslant 1 - 16^{-6},$$

которое можно изобразить в обычной позиционной записи шестью 16-ричными цифрами (или, что то же самое, двадцатью четырьмя двоичными цифрами).

Для удобства машинного представления вместо порядка p в памяти хранится характеристика $p' = p + 64$, удовлетворяющая неравенству

$$0 \leqslant p' \leqslant 127.$$

Три элемента, определяющие числа типа E: знак, характеристика p′ и мантисса m располагаются в машинном слове по схеме, представляемой на рис. 13. Знак числа записывается в разряде

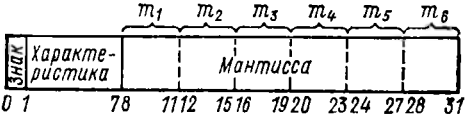


Рис. 13 Схема расположения в машинном слове короткого числа с плавающей точкой

0 по обычной системе: 0 изображает плюс, а 1 — **знак минус.** Каждая из 16-ричных цифр m_i мантиссы m занимает **четыре** бита в трех последних байтах слова. Число 0 изображается **нулями** во всех 32 битах.

Очевидно, такое представление числа неоднозначно. **Например,** число 10 можно представить в форме E шестью **способами:**

$p = 1 \qquad m = .A00000$

$p = 2 \qquad m = .0A0000$

. . . . . .

$p = 6 \qquad m = .00000A$

Представление станет однозначным, если его нормализовать, т. е. наложить на мантиссу более ограничительное условие

$$16^{-1} \leqslant m \leqslant 1 - 16^{-6}. \qquad (1)$$

Нормализованная форма — самая выгодная с точки зрения точности представления, потому что мантисса ее содержит наибольшее количество значащих цифр. Так как не всякое число типа E можно нормализовать, условие (1) несколько сужает множество представимых чисел. К нему относятся такие числа типа E, абсолютная величина которых удовлетворяет неравенствам:

$$2{,}4 \cdot 10^{-78} \doteq 16^{-65} \leqslant \mathrm{abs}\,(x) \leqslant (1 - 16^{-6}) \cdot 16^{63} \doteq 7{,}2 \cdot 10^{75}. \quad (2)$$

Нетрудно показать, что для любого рационального числа y из диапазона (2) можно указать нормализованное число x типа E, такое, что

$$\frac{\mathrm{abs}\,(y - x)}{y} \leqslant \frac{2^{-1} \cdot 16^{-6}}{16^{-1}} = 2^{-21} \doteq 10^{-6{,}4}.$$

Это значит, что любое рациональное число из диапазона (2) можно аппроксимировать нормализованным числом типа E с относительной погрешностью, не превосходящей 2^{-21}.

Арифметические операции над числами типа E, как правило, выводят за пределы этого множества. Поэтому машинные операции, представляющие результат также числом типа E, неизбежно требуют округлений. Относительная погрешность результатов этих операций по отношению к исходным операндам также составляет величину порядка 2^{-21}. Однако несмотря на то, что числа из диапазона (2) хорошо аппроксимируются числами типа E и машинные операции над числами типа E дают небольшую относительную погрешность, нельзя всегда быть уверенным в том, что результат последовательности машинных операций над числами x_i, близкими к соответственным числам y_i из диапазона (2), близок к значению последовательности точных арифметических операций над числами y_i. Неприятность происходит при вычитании близких по значению чисел. Существо этого эффекта, называемого пропаданием знаков, ясно из следующего примера.

Пусть x_1 и x_2 — соседние числа типа E (множество чисел типа E имеет не больше 2^{32} элементов), а рациональные числа y_1 и y_2 удовлетворяют неравенствам

$$x_1 < y_1 < \frac{x_1 + x_2}{2} < y_2 < x_2.$$

Очевидно, что x_1 и x_2 — ближайшие числа типа E к y_1 и y_2 соответственно. Нетрудно также убедиться, что разность $x_2 - x_1$ есть число типа E. Относительная погрешность d представления разности $y_2 - y_1$ ее машинным аналогом $x_2 - x_1$ есть

$$d = \frac{(x_2 - x_1) - (y_2 - y_1)}{y_2 - y_1} = \frac{x_2 - x_1}{y_2 - y_1} - 1.$$

Очевидно, что эта погрешность при достаточно малой разности $y_2 - y_1$ сколь угодно велика.

Число типа D отличается от короткого числа лишь тем, что его мантисса имеет не шесть, а четырнадцать 16-ричных цифр. Расположение числа в памяти приведено на рис. 14. Диапазон изменения нормализованных чисел типа D почти тот же, что и у чисел типа E:

$$16^{-65} \leqslant \mathrm{abs}(x) \leqslant 16^{63}(1 - 16^{-14}),$$

а относительная погрешность аппроксимации рационального числа из этого диапазона значительно меньше и составляет

$$2^{-53} \doteq 10^{-16}.$$

Числа типа E или D могут храниться также в четырех регистрах с плавающей точкой, имеющих номера 0, 2, 4, 6. Каждый регистр имеет 64 бита, и число типа D располагается в нем по схеме рис. 14. Число типа E занимает левую половину регистра.

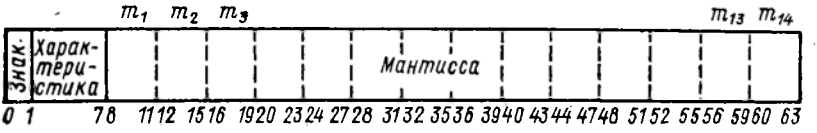


Рис. 14. Схема расположения в двойном машинном слове длинного числа с плавающей точкой

Над числами с плавающей точкой пердусмотрены 12 машинных операций: загрузка, выгрузка, сравнение, сложение, вычитание, умножение, деление, загрузка положительная, загрузка отрицательная, загрузка дополнения, загрузка с проверкой, деление пополам (Halve). Каждая операция имеет от двух до четырех модификаций, связанных с форматом команды и типом операндов. Название большинства операций на языке ассемблера состоит из одной-двух букв английского названия соответствующего действия, буквы E или D, указывающей тип операндов, и буквы R, если команда имеет формат RR.

Команды формата RR вырабатывают результат по следующей схеме:

$$R_1 := R_1 * R_2,$$

где символ * означает любую операцию, реализуемую командами формата RR. Все команды формата RX, кроме выгрузок STD и STE, действуют по схеме

$$R_1 := R_1 * D_2(X_2, B_2).$$

При выполнении команд арифметики с плавающей точкой могут произойти следующие программные прерывания:

превышение адреса;

нарушение спецификации (регистр с номером, отличным от 0, 2, 4, 6, адрес операнда в памяти не кратен 4 или 8);

защита памяти (в командах STD или STE);
переполнение порядка (порядок результата больше 63);
деление на число с нулевой мантиссой;
исчезновение мантиссы (при сложении или вычитании у результата получилась нулевая мантисса);
исчезновение порядка (порядок результата меньше числа —64).

Две последние причины могут быть замаскированы командой переключения состояния. В этом случае в качестве результата выдается нулевой код.

Таблица 7

Формат RR		Формат RX		Нормализация	Код условия	Описание операции	Время, мкс	
Длинные операнды	Короткие операнды	Длинные операнды	Короткие операнды				RR	RX
LDR	LER	LD	LE			Загрузка регистра	1,01	6,09
		STD	STE			Выгрузка в память		6,50
CDR	CER	CD	CE		да	Сравнение	4,67	7,52
ADR	AER	AD	AE	да	да	Сложение с нормализацией	5,59	8,83
AWR	AUR	AW	AU		да	Сложение без нормализации	5,28	8,53
SDR	SER	SD	SE	да	да	Вычитание с нормализацией	6,91	9,74
SWR	SUR	SW	SU		да	Вычитание без нормализации	5,48	8,32
MDR	MER	MD	ME	да		Умножение	32,89	35,74
DDR	DER	DD	DE	да		Деление	49,03	52,70
LPDR	LPER				да	Загрузка положительного	1,22	
LNDR	LNER				да	Загрузка отрицательного	1,22	
LCDR	LCER				да	Загрузка с переменой знака	1,22	
LTDR	LTER				да	Загрузка с проверкой	1,22	
HDR	HER					Деление на два	3,25	

В табл. 7 приведены мнемонические коды всех операций с плавающей точкой. Каждая строка содержит модификации одной операции. В колонке таблицы, озаглавленной: формат RX, слева указаны операции над числами типа D, а справа — операции над числами типа E. Такую же структуру имеет колонка с

заголовком: формат RR. В колонке «нормализация» словом «да» отмечены те операции, которые всегда вырабатывают нормализованный результат, а в колонке «код условия»— те, которые изменяют код условия CC. Последний вырабатывается по тому же правилу, что и в соответствующих операциях над целыми числами.

Команда сложения в строке 5 отличается от команды сложения в строке 4 лишь тем, что после выравнивания порядков и сложения мантисс результат нормализуется лишь в том случае, если сумма мантисс оказалась больше единицы. Аналогичным образом различаются команды вычитания в строках 6 и 7.

Рассмотрим несколько примеров на действия с двоичными числами.

Пример 3. Пусть в машинном слове H хранится короткое положительное двоичное целое h. Нужно это число перевести в форму E и поместить в машинное слово F.

Очевидно, что число h можно представить в форме E так:

$$h = 16^6 \left(\frac{h}{16^6}\right), \qquad (3)$$

т. е. с порядком $p = 6$ и мантиссой m, у которой старшие 8 двоичных разрядов суть нули, а 16 младших совпадают с кодом в полуслове H. Представление (3) не нормализованное, так как по крайней мере две старшие 16-ричные цифры m суть нули. Чтобы его нормализовать, достаточно выполнить операцию AE сложения с нулем. Перечисленные действия можно выполнять командами

```
      LH      0, H     R0 := H
      ST      0, F     запись в F ненормализованной ман-
*                      тиссы
      MVI     F, 70    запись характеристики в старший
*                      байт F
      SER     0, 0     очистка плавающего регистра 0
      AE      0, F     нормализованный результат в пла-
*                      вающем регистре 0
      STE     0, F     запись результата в слово F
```

В этой программе мы воспользовались командой MVI из группы логических операций (см. следующую главу), которая восьмибитовый код непосредственного операнда переносит в байт с адресом, указанным на поле первого операнда.

Аналогичное преобразование длинного целого числа в форму E может потребовать округления.

Пример 4. Рассмотрим обратную задачу: выделить из числа типа E, хранящегося в машинном слове W, целую часть и записать ее в виде двоичного целого в слово F. Обозначив через

w число типа E, ограничимся рассмотрением того случая, когда

$$0 \leqslant \text{entier}(w) \leqslant 2^{24} - 1.$$

Если мы обозначим через p 16-ричный порядок w, то положение точки, отделяющей в мантиссе m 16-ричные разряды целой части w от разрядов дробной части, изображается на рис. 15. Для решения задачи достаточно убрать из мантиссы младшие 6—p 16-ричных разрядов (или, что то же самое, младшие 24—4p двоичных), а оставшиеся 4p двоичных разрядов поместить в младшую часть слова F. Эти действия можно реализовать следующими командами:

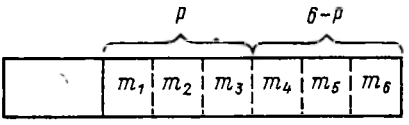


Рис. 15. Целая и дробная части

```
        LE      0, W        загрузка W в плавающий
*                           регистр 0
        AU      0, CONST    сдвиг мантиссы вправо
*                           на 24 — 4p битов
        STE     0, F        целая часть в плавающей
*                           ненормализованной
*                           форме
        MVI     F, 0        стирание характеристики
*                           в F
                . . .
CONST   DS      0F
        DC      X'46000000'
```

В слове CONST находится код числа типа E с характеристикой 70 (т. е. с порядком 6) и нулевой мантиссой. При сложении w с CONST происходит выравнивание порядков и сложение мантисс. Так как порядок p меньше шести, мантисса m сдвигается вправо на 6—p 16-ричных цифр и складывается с нулевой мантиссой CONST. По команде AU нормализация не производится и в старшей части регистра 0 остаются только нужные нам целые разряды мантиссы. Команда STE переносит код из старшей части регистра 0 в слово F, а команда MVI очищает старший байт слова F.

Пример 5. Предположим, что нам нужно преобразовать число x из десятичной формы с плавающей точкой

$$x = m \cdot 10^n,$$

где m и n — десятичные целые числа, в двоичную форму E. Преобразования такого рода обычно производятся над входными данными программы, осуществляющей вычисления с плавающей точкой.

Ограничимся случаем, когда

$$0 \leqslant m \leqslant 2^{24},$$

а n — двузначное число, причем такое, что x принадлежит диапазону (2). Алгоритм преобразования очень прост: он заключается в переводе целого m в форму E и в умножении его на число 10 (или 10^{-1}) в форме E столько раз, сколько единиц составляет модуль n. Для упрощения программы предположим, что число m в упакованной форме занимает двойное машинное слово M (т. е. изображение m содержит 15 цифр, из которых по крайней мере 7 старших равны нулю), модуль n в упакованной форме занимает двухбайтное поле N, а sign(n) в упакованной форме расположен в байте S.

```
        CVB     0, M        перевод m в двоичную
*                           форму
        ST      0, W        мантисса m в слове W
        MVI     W, 70       число m в форме E в
*                           слове W
        CP      S(1), =P'0' определение знака n
        BE      OUT         в конец при sign(n) = 0
        LE      0, W        загрузка числа m в пла-
*                           вающий регистр 0
        LE      2, =E'10'   загрузка в плавающий
*                           регистр 2 числа 10 в
*                           форме E
        BH      LOOP        обход следующей ко-
*                           манды при sign(n) = 1
        LE      2, =E'E-1'  загрузка в плавающий
*                           регистр 2 числа 10^-1
LOOP    MER     0, 2        умножение m на 10 или
*                           0. 1
        SP      N(2), =P'1' уменьшения модуля n
        CP      N(2), =P'0' проверка конца
        BNE     LOOP        на повторение умноже-
*                           ния
        STE     0, W        выгрузка результата
OUT     ....                продолжение про-
*                           граммы
```

Заметим, что при sign(n) = 0 результат в слове W окажется не нормализованным.

Пример 6. Рассмотрим обратную задачу. Пусть машинное слово W содержит полжительное число w типа E. Нужно найти представление w в виде

$$w \doteq m \cdot 10^n,$$

где m и n — десятичные целые числа.

Воспользуемся следующим алгоритмом. Пусть число w имеет представление

$$w = 16^p \cdot q.$$

Если $p = 6$, то w совпадает со своей целой частью. Это значит, что $n = 0$, а m есть десятичный эквивалент двоичного целого числа, определяемого кодом мантиссы q.

Если $p > 6$, будем умножать w последовательно на .1 до тех пор, пока после n-го умножения порядок p не станет равным 6. Это значит, что

$$w \cdot 10^{-n} = m,$$

т. е.

$$w = m \cdot 10^n.$$

Если же $p < 6$, аналогичным образом будем увеличивать порядок p до 6, умножая w на число 10 в форме E.

В следующей программе предполагается, что M — двойное машинное слово, N — поле из двух байтов, а S — байт для хранения числа +1 или −1, имеющего смысл знака n.

```
        LE      0, W        загрузка w в плаваю
*                           щий регистр 0
        ZAP     N, =P'0'    N := 0
        CLI     W, 70       P' = 70?
        BE      OUT         в конец при p = 6
        BL      MIN         при p < 6
PLUS    LE      2, =E'.1'   загрузка числа .1 в пла-
*                           вающий регистр 2,
        ZAP     S, =P'1'    если n имеет знак плюс
        B       LOOP
MIN     LE      2, =E'10'   загрузка числа 10 в пла-
*                           вающий регистр 2,
        ZAP     S, =P'-1'   если n имеет знак ми-
*                           нус
LOOP    MER     0, 2        умножение на 10 или
*                           на .1
        STE     0, W        промежуточный резуль-
*                           тат в W
        AP      N, S        n := n ± 1
        CLI     W, 70       P = 70?
        BNE     LOOP
OUT     MVI     W, 0        стирание характеристи-
*                           ки в W
        L       0, W        загрузка q в общий ре-
*                           гистр 0
        CVD     0, M        десятичный эквивалент
*                           q в двойном слове M
```

В этой программе использована логическая операция CLI, вырабатывающая в СС код сравнения байта в памяти, указанного адресом на поле первого операнда и восьмибитового кода, расположенного на поле второго операнда.

Алгоритмы примеров 5 и 6 обладают тем недостатком, что при больших по модулю n требуют многократных умножений с плавающей точкой, внося тем самым в результат ошибки округления.

§ 3.3. Организация циклов

Циклом в программе называется группа операций, которые при исполнении программы повторяются несколько раз подряд. На АЛГОЛ-60 и на других алгоритмических языках циклы могут описываться либо специальными операторами, либо с помощью условных операторов, включающих в себя операторы перехода.

С точки зрения техники программирования на машинном языке циклы удобно разделить на две группы: итерационные и циклы с параметром. К первой мы отнесем все такие циклы, которые на языке ПАСКАЛЬ (см. [11, с. 72]) описываются операторами видов:

repeat ⟨оператор⟩ **until** ⟨логическое выражение⟩ (4)

или

while ⟨логическое выражение⟩ **do** ⟨оператор⟩ (5)

По существу форма (4) отличается от (5) лишь тем, что в ней ⟨оператор⟩, называемый обычно телом цикла, выполняется по крайней мере один раз, в то время как в (5) тело цикла может не выполняться ни одного раза. В языке АЛГОЛ-60 нет специального оператора для описания итерационного цикла. На нем действия, эквивалентные соответственно (4) и (5), могут быть описаны последовательностями операторов

ТЕЛО: ⟨оператор⟩; (6)

if ¬⟨логическое выражение⟩ **then go to** ТЕЛО

и

go to ПРОВЕРКА;

ТЕЛО: ⟨оператор⟩; (7)

ПРОВЕРКА: **if** ⟨логическое выражение⟩ **then go to** ТЕЛО

Цикл с парметром на АЛГОЛ-60 описывается оператором цикла с элементом типа арифметической прогрессии

for i := a **step** b **until** c **do** ⟨оператор⟩ (8)

Говоря об этих циклах в дальнейшем, мы будем предполагать, что a, b и c — целые числа или переменные, не меняющие своих значений (так же, как и параметр i) при исполнении оператора, составляющего тело цикла. Заметим, что наша классификация, с одной стороны, условна, так как каждую из конструкций (6)—(8) можно свести к любой из двух остальных, и, с другой стороны, неполна: встречаются циклы, которые по существу дела нельзя отнести ни к одной из двух названных групп.

Циклы присутствуют во всякой мало-мальски сложной программе и подавляющая часть машинного времени, требующегося для исполнения программы, тратится на повторение команд, входящих в состав циклов. Поэтому очень важно уметь очистить циклы от всего лишнего и организовать управление ими оптимальным образом.

Что касается итерационных циклов, то, если отвлечься от структуры их тел, о которой в общем ничего сказать нельзя, управление ими сводится к экономной реализации условных операторов вида

if ⟨логическое выражение⟩ **then go to** L

Для этого нужно уметь составить оптимальную порграмму вычисления значения логического выражения в регистре CC и воспользоваться командой перехода с некоторой маской. О программировании простейших логических выражений говорилось в предыдущей главе. В более общем случае этот вопрос рассматривается в § 4.2. Здесь же мы только добавим, что кроме команды

BC M, $D_2(X_2, B_2)$ (9)

формата RX и соответствующих ей псевдокоманд можно воспользоваться командой

BCR M, R_2 (10)

формата RR, в которой маска M имеет точно тот же смысл, что и в (9), а $R_2 \neq 0$ означает номер общего регистра, в котором содержится адрес перехода. Команде (10) соответствует только одна псевдокоманда

BR R_2

безусловного перехода, которую ассемблер заменяет на

BCR 15, R_2

На этом мы оставим вопрос о программировании итерационных циклов и все внимание обратим на структуру циклов с параметром, для управления которыми в системе ЕС ЭВМ

есть четыре специальные команды перехода. Познакомимся с этими командами.

Переход по счетчику (Branch on Counter) формата RX в явной форме ассемблера имеет вид

BCT R_1, $D_2(X_2, B_2)$

R_1 — номер общего регистра, который играет роль счетчика, а $D_2(X_2, B_2)$ определяет адрес перехода. Действие этой команды в алголоподобных обозначениях можно описать так:

$$R_1 := R_1 - 1; \quad \textbf{if } R_1 \neq 0 \textbf{ then go to } D_2(X_2, B_2).$$

В этой команде содержимое общего регистра R_1 рассматривается как дополнительный код двоичного длинного числа. Например, выполнение последовательности из двух команд:

SR R, R
BCT R, *

сведется к очистке регистра R, а затем к выполнению второй команды 2^{32} раз подряд! Действительно, сначала в регистре R будут за счет вычитания единицы получаться отрицательные числа: $-1, -2, \ldots, -2^{31}$. После следующего вычитания произойдет отрицательное переполнение (на которое схема прерываний не реагирует), и в регистре R образуется код положительного числа $2^{31} - 1$, которое затем будет постепенно уменьшаться до единицы. Только тогда команда BCT R, * передаст управление следующей команде, оставив ноль в регистре R.

Переход по счетчику формата RR

BCTR R_1, R_2

отличается от только что рассмотренной команды лишь тем, что адрес перехода хранится в общем регистре $R_2 \neq 0$. Если в качестве R_2 употребляется общий регистр 0, переход к следующей по порядку расположения команде происходит независимо от значения, образовавшегося в регистре R_1. Например, команда

BCTR R, 0

действует точно так же, как команда

SH R, =H'1'

но выполняется быстрее и не требует хранения числа 1 в оперативной памяти. Идентифкатор R означает номер любого общего регистра, в том числе и регистра 0.

Переход по индексу меньшему или равному (Branch on indeX Low or Equal) формата RS в явной форме ассемблера имеет вид

BXLE R_1, R_3, $D_2(B_2)$

R_1 — номер общего регистра, содержащего парметр цикла (индекс), R_3 — номер общего регистра, хранящего шаг изменения параметра;
R_3, как правило,— четный номер. Действие команды описывается двумя следующими операторами:

$$R_1 := R_1 + R_3; \quad \textbf{if } R_1 \leqslant \overline{R_3} \textbf{ then go to } D_2(B_2)$$

Номер R_3 общего регистра определяется по правилу:

$$\overline{R_3} = \begin{cases} R_3, & R_3 \text{ нечетно}, \\ R_3 + 1, & R_3 \text{ четно}. \end{cases}$$

Например, команда

BXLE 1, 2, 0(12)

имеет четыре операнда: параметр в регистре 1, шаг его изменения в регистре 2, конечное значение параметра в регистре 3 и адрес перехода в регистре 12. Параметр, шаг и конечное значение — длинные двоичные числа любого знака. Схема прерываний процессора не реагирует на переполнение, которое может произойти при изменении параметра.

Переход по индексу бо́льшему (Branch on indeX High) формата RS в явной форме ассемблера имеет вид

BXH R_1, R_3, $D_2(B_2)$

Выполнение команды описывается операторами

$$R_1 := R_1 + R_3; \quad \textbf{if } R_1 > \overline{R_3} \textbf{ then go to } D_2(B_2),$$

где $\overline{R_3}$ имеет тот же смысл, что в предыдущей команде BXLE.

Вернемся теперь к циклу с параметром, описываемому оператором (8). Эквивалентом этого оператора может служить последовательность следующих пяти операторов:

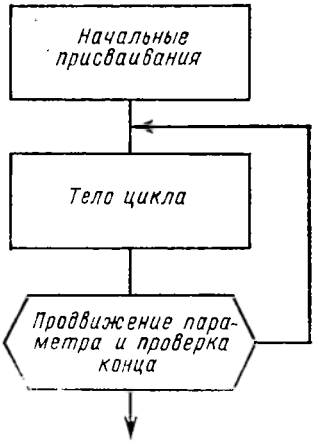
НАЧАЛЬНОЕ ПРИСВАИВАНИЕ: i := a;
go to ПРОВЕРКА;
ТЕЛО: ⟨оператор⟩; (11)
ПРОДВИЖЕНИЕ ПАРАМЕТРА: i := i + b
ПРОВЕРКА: **if** (i − c) × sign(b) ⩽ 0 **then go to** ТЕЛО

Если значения a, b и c таковы, что переход к ТЕЛО исполняется по крайней мере один раз, оператор

go to ПРОВЕРКА

можно опустить. Во всех дальнейших случаях, не делая особых оговорок, мы будем считать, что такая ситуация имеет место.

Тогда программу цикла с параметром можно изобразить следующей блок-схемой:



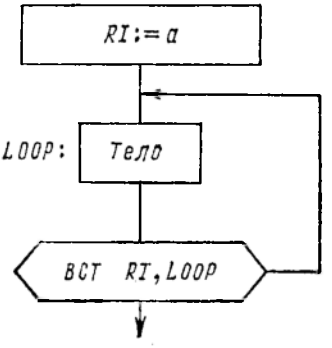
(12)

В этой схеме первый блок представляет группу команд, исполняемых один раз. Остальные два блока исполняются, вообще говоря, многократно. Об организации второго блока в общем случае трудно дать какие-либо конкретные рекомендации. Нужно постараться вынести из него в первую часть все те операции, которые достаточно выполнить один раз. Смысл таких действий, называемых чисткой циклов (см. [12, с. 99]), мы поясним дальше на примерах. Блоки же первый и третий в широких классах случаев имеют более или менее стандартную структуру. Рассмотрим эти случаи, считая величины a, b и c двоичными целыми числами или переменными.

Предположим, что

$$a > 0, \quad b = -1, \quad c = 1$$

и изберем для хранения параметра i цикла общий регистр, обозначенный идентификатором RI. Тогда схема (12) сведется к



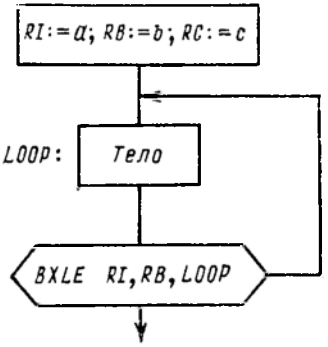
(13)

Следует твердо помнить, что тело цикла в этом случае выполняется ровно a раз.

Предположим, что шаг положителен, а начальное значение параметра не превосходит конечного

$$b > 0, \quad a \leqslant c.$$

Изберем для хранения параметра цикла общий регистр RI, для шага b выберем общий регистр с четным номером RB, а конечное значение поместим в регистр со следующим нечетным номером RC. Тогда схема (12) превратится в



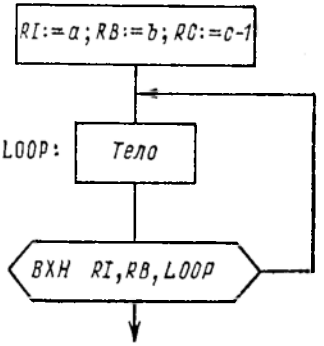
(14)

Отметим, что (14) соответствует (11) и в тех случаях, когда команды, входящие в состав тела цикла, меняют значения переменных i, b и c в регистрах соответственно RI, RB, RC при условии, что шаг b остается строго положительным.

Предположим, наконец, что шаг b отрицателен, а конечное значение параметра не превосходит начального

$$b < 0, \quad a \geqslant c.$$

Обозначим, как в предыдущем случае, идентификаторами RI, RB и RC общие регистры для управления циклом, считая, что RB — четный номер, а RC — следующий нечетный. Тогда схема (12) превратится в следующую:



(15)

В качестве ограничивающего значения в регистре RC можно взять любое число z, удовлетворяющее неравенствам

$$x + b \leqslant z < x,$$

где x означает последнее значение параметра i, при котором, согласно алгоритму (11), выполняется тело цикла. Подчеркнем, что если мы поместим в регистр RC число c, количество повторений тела цикла может оказаться на единицу меньшим того, которое предписывается алгоритмом (11).

Схемы (13)—(15), представляющие частные случаи (12), сравнительно редко удается применить в чистом виде. Дело в том, что параметр цикла обычно фигурирует в разных формах в различных командах тела цикла. Поэтому для его продвижения приходится применять дополнительные команды, которые естественно включить в третий блок рассматриваемого семейства схем. По той же причине дополнительные операции могут потребоваться в блоках начального присваивания.

Пример 7. Пусть в общем регистре 0 нужно вычислить в виде двоичного целого сумму кубов всех нечетных чисел от 1 до 25. На АЛГОЛ-60 решение можно описать так:

R0 := 1; **for** i := 3 **step** 2 **until** 25 **do** R0 := R0 + i ↑ 3

Для реализации оператора цикла воспользуемся схемой (14). Параметр цикла i будем хранить в регистре 1, а для шага 2 и конечного значения 25 возьмем регистры соответственно 14 и 15. Кроме того, нам понадобятся для образования i↑3 еще два регистра: четный и следующий нечетный. Пусть это будут регистры 2 и 3.

```
        LA      0, 1        R0 := 1
* начальные присваивания
        LA      1, 3        i := 3
        LA      14, 2       шаг изменения параметра i
        LA      15, 25      конечное значение параметра i
* тело цикла
LOOP    LR      3, 1        R3 := i
        MR      2, 3        R2, 3 := i ↑ 2
        MR      2, 1        R2, 3 := i ↑ 3
        AR      0, 3        R0 := R0 + i ↑ 3
* продвижение параметра и проверка конца
        BXLE    1, 14, LOOP
```

Употребление операторов цикла с параметром в задачах, подобных рассмотренной в примере 7, на практике имеет место сравнительно редко. Главная область применения этих операторов — действия с элементами прямоугольных массивов.

§ 3.4. Операции с прямоугольными массивами

Пусть a — одномерный массив (вектор) в смысле АЛГОЛ-60 с описанием

integer array a [l : u]

элементами которого являются целые числа длины s, а границы l и u известны в момент написания программы. Естественно располагать элементы этого массива в оперативной памяти вплотную друг к другу в порядке возрастания индексов, как

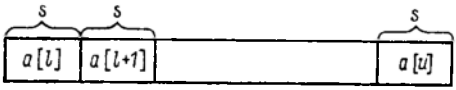


Рис. 16. Расположение в памяти элементов одномерного массива

показано на рис. 16. Тогда, если мы обозначим через A адрес первого байта элемента a[l], то очевидно, что адрес первого байта i-го элемента определится по формуле

$$A + (i - l) \times s. \tag{16}$$

Обозначим через A0 постоянную часть выражения (16)

$$A0 = A - l \times s. \tag{17}$$

Тогда адрес элемента a[i], являющийся линейной функцией индекса i, представляется в виде

$$\text{адрес } a[i] = A0 + i \times s. \tag{18}$$

Значение слагаемого A0 может быть определено до исполнения программы. Из формулы (18) следует, что A0 есть адрес элемента a[0]. Если границы l и u удовлетворяют неравенствам

$$l \leqslant 0 \leqslant u,$$

то такой элемент есть в нашем массиве и поэтому A0 не выходит за пределы участка памяти, занимаемого нашим массивом. Если же 0 не принадлежит замкнутому промежутку [l, u], адрес A0 лежит вне участка, изображенного на рис. 16 *.

Предположим, что i является параметром некоторого цикла. Каким образом можно организовать доступ к элементам a[i]? Если соответствующая машинная команда имеет формат RX, то можно постоянную компоненту A0 адреса a[i] задать с помощью смещения D_2 и базисного регистра B_2, а слагаемое i × s вычислять в индексном регистре X_2. Если же адрес опе-

* Если это имеет место и мы указываем адрес нулевой компоненты переместимым выражением в команде формата RX (см., например, команду LH 2, A-10(1) в примере 8), мы должны при резервировании памяти для массива позаботиться о том, чтобы этот адрес отличался от адреса в базисном регистре программы на положительное смещение, не превосходящее 4095.

ранда формируется с помощью только одного регистра (случаи формата RS, SS и SI), адрес a[i] приходится вычислять в базисном регистре команды. Рассмотрим два простых примера.

Пример 8. Пусть

integer array a, b [5 : 25]

— массивы из коротких двоичных чисел. Нужно в слове с адресом C образовать скалярное произведение векторов a и b. На АЛГОЛ-60 необходимые действия можно описать так:

R := 0; **for** i := 5 **step** 1 **until** 25 **do** R := R + a [i] × b [i]; C := R

Для того чтобы по возможности приблизить запись на АЛГОЛ-60 к машинной программе, мы использовали для накопления суммы произведений переменную R, которая будет реализована в одном из общих регистров.

Предполагая, что память для массивов a и b резервирована командами ассемблера

```
A       DS      21H
B       DS      21H
```

по формуле (16) получим

адрес a[i] = A − 10 + 2 × i
адрес b[i] = B − 10 + 2 × i

Так как в нашем случае параметр цикла i входит только в выражения для адресов a[i] и b[i] и притом с множителем 2, выгодно считать параметром цикла величину 2 × i, изменяющуюся с шагом 2 от 10 до 50.

Займем следующие общие регистры:

0 — для накопления суммы,
1 — для параметра 2 × i,
14 — для шага 2,
15 — для конечного значения 50,
2 — для получения очередного произведения.

Для управления циклом воспользуемся схемой (14) с командой BXLE.

```
        SR      0, 0            R := 0
* начальные присваивания
        LA      1, 10           начальное значение 2 × i
        LA      14, 2           шаг 2
        LA      15, 50          конечное значение 2 × i
* тело цикла
LOOP    LH      2, A − 10 (1)   R2 := a [i]
        MH      2, B − 10 (1)   R2 := a [i] × b [i]
        AR      0, 2            R := R + a [i] × b [i]
```

```
* проверка конца и продвижение параметра
          BXLE     1, 14, LOOP
* действие после окончания цикла
          ST       0, C
```

Рассмотрим другой вариант решения той же задачи, организовав управление циклом с помощью схемы (13). Оставив за регистрами 0, 1, 2 прежнюю роль, используем регистр 15 в качестве счетчика в команде BCT. Запись алгоритма на АЛГОЛе можно преобразовать следующим образом:

R := 0; **for** j := 21 **step** −1 **until** 1 **do** R := R + a[j + 4] × b[j + 4]; C := R.

Теперь

$$\text{адрес } a[j+4] = A - 2 + 2 \times j,$$
$$\text{адрес } b[j+4] = B - 2 + 2 \times j.$$

Считая параметром цикла в машинной программе величину 2 × j, напишем ее в виде

```
          SR       0, 0          R := 0
* начальное присваивание
          LA       15, 42        начальное значение 2 × j
* тело цикла
LOOP      LH       2, A — 2 (15) R2 := a[j + 4]
          MH       2, B — 2 (15) R2 := a[j + 4] × b[j + 4]
          AR       0, 2          R := R+a[j+4]×b[j+4
* продвижение параметра и проверка конца
          BCTR     15, 0         R15 := R15 — 1
          BCT      15, LOOP
* действия после окончания цикла
          ST       0, C          C := R
```

Возникает естественный вопрос: какой вариант лучше? Различные варианты программ, производящих одни и те же действия, сравниваются по количеству ресурсов, которые они требуют у вычислительной системы для своего исполнения. В нашем случае такими ресурсами являются: оперативная память, общие регистры, время центрального процессора. Второй вариант короче на одну команду и не занимает регистр 14. Ясно, что он несколько выгоднее по двум первым параметрам.

Однако циклическая часть второй программы на одну команду длиннее: она включает команды BCTR и BCT вместо одной BXLE в первом варианте. А так как наш цикл повто-

ряется 21 раз, разность во времени исполнения вариантов существенно зависит от того, насколько время, требуемое командой BXLE, отличается от суммарного времени для BCTR и BCT. В разных моделях системы ЕС ЭВМ на одну и ту же команду тратится различное время. Это время зависит также (причем в разных моделях по разному) от программного контекста, поэтому точный подсчет представляет значительные затруднения. Для сравнения вариантов обычно пользуются средними данными, приведенными в характеристике каждого процессора. Например, для модели 1033 среднее время трех названных выше команд составляет соответственно 3,05, 3,25 и 3,65 мкс. Нетрудно подсчитать, что на этой машине время исполнения второго варианта на 10% больше времени первого варианта, т. е. по третьему параметру более выгодным оказывается первый вариант.

Пример 9. Пусть векторы

integer array a, b[5 : 25]

состоят из десятичных упакованных чисел длиной 3. Нам нужно на поле S длиной 6 получить скалярное произведение векторов в той же форме. Предположим, что места в памяти для переменных программы резервированы командами ассемблера

```
A     DS    21PL3
B     DS    21PL3
S     DS    PL6
W     DS    PL6       поле для умножения
```

Опишем необходимые действия следующими операторами:

$$S := 0;\ \textbf{for}\ i := 5\ \textbf{step}\ 1\ \textbf{until}\ 25\ \textbf{do}$$
$$\textbf{begin}\ W := a[i];\quad W := W \times b[i];\quad S := S + W\ \textbf{end}$$

Адреса элементов a[i] и b[i] на основании (18) зависят от i следующим образом:

$$\text{адрес}\ a[i] = A - 15 + 3 \times i,$$
$$\text{адрес}\ b[i] = B - 15 + 3 \times i.$$

Так как соответствующие команды имеют формат SS, эти адреса следует вычислять в базисных регистрах операндов. В качестве параметра цикла машинной программы здесь удобно принять адрес переменной a[i], а для вычисления адреса b[i] воспользоваться тем, что разность между этими адресами не зависит от i:

$$\text{адрес}\ b[i] = B - A + \text{адрес}\ a[i]$$

Для управления циклом изберем схему (14) с командой BXLE, заняв для параметра, его шага и конечного значения регистры 14, 15 и 1:

```
        ZAP     S, =P'0'        S := 0
* начальные присваивания
        LA      14, A           начальное значение
*                               адреса a[i]
        LA      15, 3           шаг 3
        LA      1, A+60         конечное значение
*                               адреса a[i]
* тело цикла
LOOP    ZAP     W, 0(3, 14)     W := a[i]
        MP      W, B-A(3, 14)   W := W × b[i]
        AP      S, W
* продвижение параметра и проверка конца
        BXLE    14, 1, LOOP
```

Выражение, расположенное в позиции смещения второго операнда команды MP, является абсолютным выражением ассемблера, так как его значение не зависит от места расположения в оперативной памяти машинной программы. Для

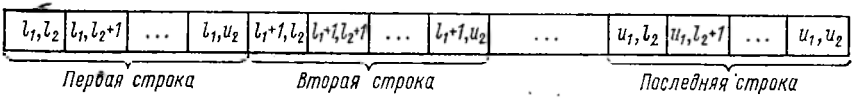


Рис. 17. Схема расположения в памяти элементов двумерного массива

того чтобы оно было положительным, необходимо, чтобы команда DS, описывающая имя B, располагалась во входной программе после команды, описывающей имя A.

Рассмотрим теперь двумерные массивы (матрицы). Пусть

array $a[l_1 : u_1, l_2 : u_2]$

— описание такого массива. Как расположить его элементы в оперативной памяти машины, имеющей линейную структуру? Очевидно, предпочтительно такое расположение, когда элементы располагаются в таком порядке, в каком они подлежат обработке. А так как обычный порядок обращения к элементам матрицы — вдоль по рядам (т. е. по строчкам или столбцам), то обычно применяется одно из двух естественных размещений: по строчкам (т. е. в лексикографическом порядке) или по столбцам. Мы во всех дальнейших примерах будем пользоваться лексикографическим расположением.

Пусть каждый элемент матрицы занимает s_2 байтов. Тогда вся матрица при плотном расположении потребует участок в $(u_1 - l_1 + 1) \times (u_2 - l_2 + 1) \times s_2$ байтов. Для одной строки необходимо $(u_2 - l_2 + 1) \times s_2$ байтов. Порядок расположения элементов приведен на рис. 17. Элементы каждой строки расположены друг за другом в порядке возрастания второго индекса, а строки расположены в порядке возрастания первого индекса. Если мы обозначим через s_1 длину участка, занятого

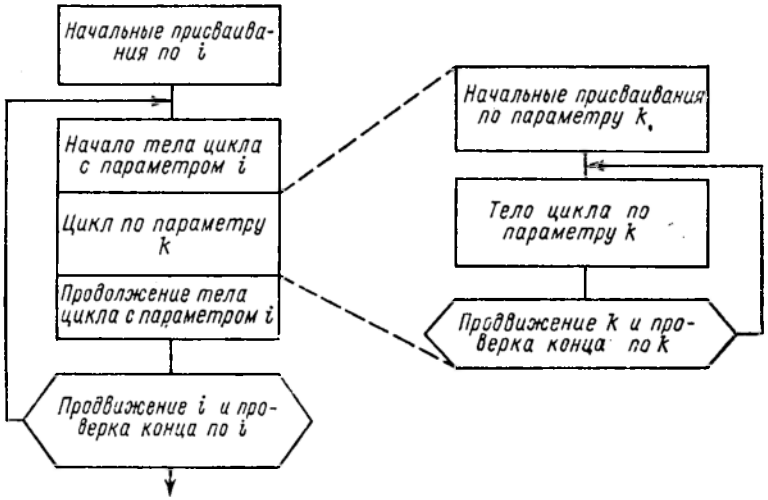


Рис. 18. Схема двойного цикла

одной строкой, а через A — адрес первого байта элемента $a[l_1, l_2]$, то очевидно, что

$$\text{адрес } a[i, k] = A + (i - l_1) \times s_1 + (k - l_2) \times s_2. \qquad (19)$$

Обозначив через A0 не зависящую от индексов компоненту адреса

$$A0 = A - l_1 \times s_1 - l_2 \times s_2,$$

получим

$$\text{адрес } a[i, k] = A0 + i \times s_1 + k \times s_2. \qquad (20)$$

Числа s_1 и s_2 называются шагами изменения адреса по первому и второму индексам. A0 так же, как и в одномерном случае, есть адрес элемента $a[0, 0]$, который на самом деле может отсутствовать.

Если компонента $a[i, k]$ является операндом команды формата RX, постоянное слагаемое адреса A0 можно выразить через смещение D_2 и базисный регистр B_2 второго операнда (см. сноску к с. 72), а зависящую от индексов часть

$$i \times s_1 + k \times s_2$$

вычислять в индексном регистре X_2 перед каждым исполнением команды. Возможны и другие варианты. Например, сумму $A0 + i \times s_1$ можно вычислять в базисном регистре B_2, а величину $k \times s_2$ — в индексном регистре X_2.

Если компонента a[i, k] является операндом команды формата SS, RS или SI, адрес $A0 + i \times s_1 + k \times s_2$ приходится каким-либо способом вычислять в базисном регистре операнда перед каждым исполнением команды.

Индексы i и k часто служат параметрами вложенных друг в друга циклов. Например, i может быть параметром цикла, тело которого содержит в качестве своей части другой цикл с параметром k. Структура двойного цикла в достаточно общем случае описывается схемой, изображенной на рис. 18.

Если параметр i пробегает n значений, а параметр k — m значений, кратность повторения команд, входящих в состав второго и третьего блоков внутреннего цикла с параметром k, составляет $n \times m$. Поэтому необходимо программировать эти блоки особенно тщательно. Все те операции, которые можно повторять с меньшей частотой, должны быть вынесены во внешние части цикла.

Пример 10. Рассмотрим задачу умножения квадратной матрицы на вектор. Пусть

```
array a[1 : 10, 1 : 10], b, c[1 : 10]
```

— массивы, элементами которых являются машинные слова. Предположим, что память для этих массивов резервирована командами:

```
A     DS    100F
B     DS     10F
C     DS     10F
```

и матрица a и вектор b обладают значениями, состоящими из чисел типа E. Нам нужно получить в векторе c результат умножения a на b. В терминах АЛГОЛа напишем нашу программу так:

```
for   i := 1   step 1 until 10 do
      begin    RF0 := 0;
               for k := 1 step 1 until 10 do
               begin RF2 := a[i, k];   RF2 := RF2 × b[k];
                     RF0 := RF0 + RF2 end;
               c[i] := RF0
      end
```

Идентификаторами RF0 и RF2 мы обозначили регистры с плавающей точкой соответственно 0 и 2. Алгоритм расчленен так, что каждый оператор присваивания можно выразить одной ма-

шинной командой. Предполагая, что элементы расположены в лексикографическом порядке, найдем

адрес a[i, k] = A − 44 + 40 × i + 4 × k
адрес b[k] = B − 4 + 4 × k
адрес c[i] = C − 4 + 4 × i

Займемся сначала обеспечением внутреннего цикла. Мы видим, что в выражения адресов a[i, k] и b[k] параметр k входит с множителем 4. Поэтому естественно изменять его во внутреннем цикле с шагом 4 в пределах от 4 до 40. Используя для управления внутренним циклом схему (14), займем для шага 4 и конечного значения 40 общие регистры 0 и 1, а величину 4 × k будем хранить в регистре 14. Так как b[k] является вторым операндом команды формата RX, его адрес может быть написан в виде

B − 4(14).

Во внутреннем цикле фигурирует также операнд b[i, k]. Слагаемое его адреса, зависящее от k, можно получить из регистра 14, указанного в качестве индексного регистра операнда. Второе слагаемое

A − 44 + 40 × i

будем формировать во внешнем цикле в базисном регистре операнда, в качестве которого выберем общий регистр 15. Тогда мы сможем указать этот операнд в команде загрузки в явной форме 0(14, 15).

Теперь остается обеспечить управление внешним циклом. Параметр i входит в два места: в адреса a[i, k] и c[i]. Зависящее от i переменное слагаемое первого адреса мы уже решили формировать в регистре 15. Для слагаемого 4 × i второго адреса займем общий регистр 2. Тогда на операнд c[i] в команде выгрузки можно сослаться в неявной форме C − 4(2). Для управления внешним циклом воспользуемся той же схемой (14) с шагом 4. Для хранения шага и конечного значения 40 можно использовать те же регистры 0 и 1, что и во внутреннем цикле.

```
* начальное присваивание по параметру i
          LA      0, 4          шаг по i и k
          LA      1, 40         конечное значение
*                               для 4×i и 4×k
          LR      2, 0          начальное значение
*                               для 4x
          LA      15, A − 4     начальное значение
*                               для A − 44 + 40×i
```

```
* начало тела цикла по параметру i
LOOPI    SER     0, 0              RF0 := 0
* начальное присваивание по параметру k
         LR      14, 0             начальное значение
*                                  для 4 × k
* тело цикла по параметру k
LOOPK    LE      2, 0(14, 15)      RF2 := a[i, k]
         ME      2, B — 4(14)      RF2 := RF2 × b[k]
         AER     0, 2              RF0 := RF0 + a[i, k] ×
*                                  × b[k]
* продвижение k и проверка конца по k
         BXLE    14, 0, LOOPK      продвижение 4 × k
*                                  в регистре 14
* продолжение тела цикла по параметру i
         STE     0, C — 4(2)       c[i] := RF0
* продвижение i и проверка конца по i
         AR      15, 1             продвижение A — 44 +
*                                  + 40 × i в регистре 15
         BXLE    2, 0, LOOPI       продвижение 4 × i в
                                   регистре 2
```

Пример 11. Пусть матрица
array a[0 : 10, 0 : 10]

состоит из машинных слов. Нам нужно транспонировать эту матрицу (т. е. взаимно обменять значения элементов, симметричных относительно главной диагонали). На АЛГОЛе нашу программу можно написать так:

```
                for i := 1 step 1 until 10 do
                for k := 1 step 1 until i do
begin           R1 := a[i, k — 1]; R2 := a[k — 1, i];
                a[i, k — 1] := R2; a[k — 1, i] := R1
end
```

В этой записи идентификаторы R1 и R2 обозначают переменные, которые в машинной программе будут представлены двумя общими регистрами, используемыми для перестановки значений симметричных элементов. Кроме того, мы сдвинули на 1 границы изменения параметра k внутреннего цикла. Это несколько упростит машинную программу, так как теперь верхняя граница внутреннего параметра совпадает с текущим значением внешнего параметра i.

Предположим, что место в памяти для нашей матрицы резервировано командой ассемблера

```
A       DS      121F
```

и что ее элементы расположены в лексикографическом порядке. По формуле (19) найдем

$$\text{адрес } a[i, k-1] = A - 4 + 44 \times i + 4 \times k$$
$$\text{адрес } a[k-1, i] = A - 44 + 4 \times i + 44 \times k$$

Так как команды загрузки и выгрузки имеют формат RX, можно для формирования каждого из адресов воспользоваться двумя регистрами. Займем следующие регистры:

регистр 1 — для $4 \times i$
регистр 2 — для $4 \times k$
регистр 14 — для $A - 4 + 44 \times i$
регистр 15 — для $A - 44 + 44 \times k$

Величины $4 \times i$ и $4 \times k$ могут служить в машинной программе параметрами внешнего и внутреннего циклов, так как непосредственно i и k нигде в программе не фигурируют. Для управления внутренним циклом воспользуемся схемой (14). Для шага 4 изменения внутреннего параметра $4 \times k$ изберем регистр 0, так как конечное значение этого параметра $4 \times i$ лежит в регистре 1. Параметр внутреннего цикла с шагом 44 находится также в регистре 15. Это необходимо учесть при начальном присваивании и при продвижении параметра внутреннего цикла. Для управления внешним циклом воспользуемся схемой (13) с командой BCT. Для счетчика займем регистр 3. Так как параметр внешнего цикла с шагом 44 должен присутствовать и в регистре 14, нужно не забыть соответствующие команды в блоках начального присваивания и продвижения параметра. Для обмена значений употребим общие регистры 4 и 5.

```
* начальные присваивания по параметру i
          LA      1, 4          начальное значение
*                               4 X i
          LA      3, 10         счетчик внешнего
*                               цикла
          LA      14, A + 40    начальное значение
*                               A - 4 + 44 X i
          LR      0, 1          шаг изменения 4 X k
*                               во внутреннем цикле
* тело внешнего цикла, начальное присваивание по k
LOOPI     LR      2, 0          начальное значение
*                               4 X k
          LA      15, A         начальное значение
*                               A - 44 + 44 X k
```

```
* тело внутреннего цикла
LOOPK    L       4, 0(14, 2)     R4 := a[i, k — 1]
         L       5, 0(15, 1)     R5 := a[k — 1, i]
         ST      5, 0(14, 2)     a[i,k — 1] := R5
         ST      4, 0(15, 1)     a[k — 1, i] := R4
* продвижение и проверка конца по k
         LA      15, 44(15)      продвижение k в ре-
*                                гистре 15
         BXLE    2, 0, LOOPK     продвижение k в ре-
*                                гистре 2
* продолжение внешнего цикла: продвижение i и проверка конца
         LA      14, 44(14)      продвижение 44 × i в
*                                регистре 14
         LA      1, 4(1)         продвижение 4 × i в
*                                регистре 1
         BCT     3, LOOPI        проверка конца по i
```

Для управления внешним циклом можно также воспользоваться схемой (14) с командой BXLE. Для этого нам нужно занять два соседних общих регистра — четный и нечетный — для шага 4 и конечного значения 40. Можно воспользоваться для этого, например, регистрами 6 и 7, освободив регистр 3. Тогда две последние команды нашей программы можно заменить одной:

```
         BXLE    1, 6, LOOPI
```

а в блоке начальных присваиваний команду

```
         LA      3, 10
```

заменить на две:

```
         LR      6, 1
         LA      7, 40
```

Циклическая часть программы сократилась на одну команду, т. е. возросла скорость ее исполнения (см. описание программы примера 8). Однако нам пришлось «заплатить» за это дополнительным общим регистром.

Пример 12. Пусть массив

integer array a[1 : 20]

заполнен десятичными упакованными числами длины 5. Нужно расположить эти числа в порядке возрастания. Воспользуемся

одним из вариантов метода пузырька

```
for i := 19 step -1 until 1 do
        begin   ind := 0;
                for k := 1 step 1 until i do
                if a[k] > a[k + 1] then begin s := a[k];
                a[k] := a[k + 1];
                a[k + 1] := s; ind := 1 end;
                if ind = 0 then do to out
        end;
out:
```

В этом алгоритме при каждом исполнении внутреннего цикла наибольшее из значений элементов a[1], a[2], ..., a[i] передвигается последовательными транспозициями на самое правое место. Переменная ind служит признаком досрочного завершения работы, если до естественного окончания внешнего цикла все элементы окажутся упорядоченными.

Пусть места для переменных программы резервированы командами

```
A       DS      20PL5
S       DS      PL5
IND     DS      PL1
```

Ясно, что

адрес a[k] = A − 5 + 5 × k.

Так как все команды, ссылающиеся на a[k] или a[k + 1], имеют формат SS, необходимо вычислять адреса элементов в базисных регистрах операндов. В рассматриваемом случае удобно в качестве параметров как внешнего, так и внутреннего циклов вместо индекса элемента использовать его машинный адрес. Для управления внутренним циклом выберем схему с командой BXLE. Адрес A − 5 + 5 × k будем хранить в регистре 2, шаг 5 — в регистре 0. Тогда конечное значение параметра внутреннего цикла, равное A − 5 + 5 × i, должно вычисляться в регистре 1. Для управления внешним циклом употребим схему (15) с командой BXH. Для текущего значения параметра — адреса A − 5 + 5 × k — нами уже избран регистр 1. Шаг −5 и уменьшенное на 1 конечное значение A−1 расположим в регистрах соответственно 14 и 15.

```
* начальное присваивание по i
                LA      1, A + 90       начальное значение
*                                       A − 5 + 5 × i
                LA      0, 5            шаг по k
                LNR     14, 0           шаг −5 по i
                LA      15, A − 1       ограничивающее зна-
*                                       чение по i
```

```
* начало тела внешнего цикла
LOOPI    ZAP     IND, =P'0'      ind := 0
* начальное присваивание по k
         LA      2, A            начальное значение
*                                A — 5 + 5 × k
* тело цикла по k
LOOPK    CP      0(5, 2), 5(5, 2)   a[k] > a[k + 1]?
         BNH     ENDK               переход, если a[k] ⩽
*                                   ⩽ a[k + 1]
         ZAP     S, 0(5, 2)         S := a[k]
         ZAP     0(5, 2), 5(5, 2)   a[k] := a[k + 1]
         ZAP     5(5, 2), S         a[k + 1] := S
         ZAP     IND, =P'1'         ind := 1
* продвижение параметра и проверка конца по k
ENDK     BXLE    2, 0, LOOPK
* продолжение внешнего цикла
         CP      IND, =P'0'      if ind = 0 then
         BE      *+8             go to out
         BXH     1, 14, LOOPI
```

До сих пор мы рассматривали только векторы, матрицы и вложенные двукратные циклы. Однако почти все сказанное естественно переносится и на более общие случаи. Пусть, например,

array $a[l_1 : u_1, l_2 : u_2, l_3 : u_3]$

— описание трехмерного массива, все элементы которого имеют длину s_3. Если мы расположим эти элементы вплотную друг к другу в лексикографическом порядке (т. е. в таком, когда элементы следуют в порядке возрастания индексов, причем самым старшим является первый, а младшим — последний), то имеют место формулы

$$\text{адрес } a[i, j, k] = A0 + s_1 \times i + s_2 \times j + s_3 \times k,$$
$$s_2 = (u_3 - l_3 + 1) \times s_3,$$
$$s_1 = (u_2 - l_2 + 1) \times s_2,$$
$$A0 = A - l_1 \times s_1 - l_2 \times s_2 - l_3 \times s_3.$$

Здесь A, как и раньше, означает адрес первого байта элемента $a[l_1, l_2, l_3]$, а A0 имеет смысл адреса элемента $a[0, 0, 0]$, которого на самом деле может и не быть.

Выражения, стоящие в позициях индексов элементов, в общем случае могут быть произвольными функциями от параметров циклов, управляющих обработкой массива. Поэтому адрес элемента может нелинейно зависеть от параметров и в самом

общем случае затруднительно дать сколько-нибудь определенные рекомендации о порядке его вычисления. Напомним только еще раз, что для экономии машинного времени нужно как можно более тщательно программировать внутренние циклы, вынося из них все лишнее в более внешние части, и максимально использовать для управления этими циклами и реализации входящих в них операций общие регистры и регистры с плавающей точкой. В случае недостатка свободных регистров можно (если это позволяет алгоритм задачи) перед входом во внутренние циклы с помощью команды STM выгружать в память содержимое группы общих регистров, использовать эти регистры для организации внутренних циклов, а по выходе из циклов командой LM восстанавливать сохраненную информацию.

В заключение раздела приведем пример фрагмента программы, содержащего обращение к элементу массива, нелинейно зависящему от параметра цикла i. Пусть

integer array $a[0:u_1]$, $b[0:u_2]$

— векторы, состоящие соответственно из машинных слов и полуслов, причем

$$\text{адрес } a[i] = A + 4 \times i, \quad \text{адрес } b[k] = B + 2 \times k$$

Пусть параметр цикла i хранится в регистре 2, а в регистр 1 нужно загрузить значение $a[i \times b[i]]$. Предполагая, что общий регистр 0 свободен, можем наш фрагмент написать так:

LR	1, 2	i в регистре 1
SLA	1, 1	$2 \times i$ в регистре 1
LH	1, B(1)	b[i] в регистре 1
MR	0, 2	$i \times b[i]$ в регистре 1
SLA	1, 2	$4 \times i \times b[i]$ в регистре 1
L	1, A(1)	$a[i \times b[i]]$ в регистре 1

§ 3.5. Динамическое распределение памяти

До сих пор мы предполагали, что границы изменения индексов в массивах известны при написании программы. Но очень часто бывает так, что автору известны только размерности массивов, а границы изменения индексов (и, следовательно, количество элементов) определяются лишь при выполнении программы. В такой ситуации мы, вообще говоря, не можем отводить определенный участок памяти для каждого массива в момент написания программы и, следовательно, лишены возможности считать шаги по индексам и адрес нулевой компоненты постоянными программы.

В описанных условиях мы находимся, например, тогда, когда переводим на машинный язык программу на АЛГОЛе, имеющую блочную структуру: при каждом входе в блок нужно отводить в памяти место для описанных в блоке массивов, а при каждом выходе освобождать занятую память, чтобы она могла быть вновь использована для хранения массивов следующего в порядке исполнения (может быть, того же самого) блока. Такая процедура называется динамическим распределением памяти. Занятие и освобождение памяти происходит по принципу стека: участок, занятый в последнюю очередь для массивов самого внутреннего блока, освобождается в первую очередь.

В настоящем разделе мы ознакомимся с методами динамического резервирования памяти и техникой программирования обращений к элементам таких (часто называемых динамическими) массивов.

Один из возможных методов резервирования может быть следующий: в конце программы, написанной на языке ассемблера, с помощью одной команды DS запрашиваем участок, достаточный для хранения всех, нужных одновременно динамических массивов программы, и используем одно машинное слово (общий регистр) для хранения указателя стека — адреса первого свободного байта нашего участка. В процессе исполнения программы при каждом входе и выходе из блока специальные команды должны изменять значение этого указателя, отделяя занятую текущими массивами часть участка от свободной, доступной для массивов следующих блоков.

Этот способ неудобен по нескольким причинам. Одной из них является тот факт, что заказанный ассемблером участок памяти становится частью объектной программы и будет зря занимать место, например, при хранении программы в библиотеке.

Другой способ заключается в резервировании и освобождении участков памяти во время исполнения машинной программы с помощью макрокоманд супервизора GETMAIN и FREEMAIN. Макрокоманда GETMAIN (выдать участок оперативной памяти) в простейшем варианте имеет два операнда:

$$\text{GETMAIN} \qquad \text{R,} \left\{ \begin{array}{l} \text{LV} = \text{десятичное число} \\ \text{LV} = (0) \end{array} \right\}$$

Фигурные скобки указывают возможные альтернативы: длина заказываемого участка (Length Value) может быть указана либо десятичным числом, либо двоичным числом в общем регистре 0. Длина участка должна быть кратна 8, в противном случае она автоматически округляется до 8 в бо́льшую сторону. Если в момент исполнения заказа система может выделить программе такой участок, она помещает адрес первого

байта этого участка, начинающегося всегда на границе двойного слова, в общий регистр 1 и продолжает исполнение программы. Если же свободного участка нет, исполнение программы прекращается.

В той точке программы, по достижении которой заказанный ранее командой GETMAIN участок памяти (или, что менее желательно, его часть) становится свободным, следует поместить макрокоманду FREEMAIN (считать свободным участок оперативной памяти), которая в простейших случаях имеет три операнда:

$$\text{FREEMAIN} \quad R, \begin{Bmatrix} LV = \text{десятичное число} \\ LV = (0) \end{Bmatrix}, \begin{Bmatrix} A = \text{переместимое выражение} \\ A = (1) \end{Bmatrix}$$

Ключевой операнд LV указывает длину освобождаемого участка, а операнд A — адрес его первого байта, задаваемый либо переместимым выражением, указывающим адрес машинного слова, которое хранит адрес начала участка, либо двоичным числом в регистре 1. Длина и адрес обязательно должны быть кратны числу 8.

Более подробные сведения о макрокомандах динамического распределения памяти можно найти в [10].

Для организации доступа к элементам динамических массивов нужно для каждого массива завести паспорт массива. Все такие паспорта обычно располагаются в статической части программы *. Возможная структура паспорта двумерного массива в принятых ранее обозначениях приведена на рис. 19.

F	H	H	H	H	H	H
A0	l_1	u_1	s_1	l_2	u_2	s_2

Рис. 19. Паспорт двумерного массива

Для слагаемого A0 отводят машинное слово, а для границ и шагов изменения адреса по индексам обычно достаточно машинных полуслов. Некоторые позиции паспорта могут заполняться при написании программы, другие (в том числе A0) вычисляются в процессе ее исполнения.

На месте каждого входа в блок должна быть написана программа заказа памяти и заполнения паспортов массивов, описанных в этом блоке.

---

* Статической частью программы называется совокупность тех ее элементов: команд, постоянных, переменных, память для которых отводится во время написания программы. Если программа имеет блочную структуру, целесообразно динамически отводить память только для ее массивов, а для простых переменных, в том числе и тех, которые описаны во внутренних блоках, резервировать место командами DS или DC при написании программы.

В том месте программы блока, где производится ссылка на массив, должна быть написана программа доступа к элементам массива.

В месте выхода из блока нужно написать программу освобождения памяти, занятой массивами блока.

Покажем на простом примере, как могут выглядеть названные программы. Предположим, что входная программа есть

```
. . .
      begin integer array a[1 : u];
. . .
      z := a[3 × (i + j) + 2];
. . .
      end;
. . .
```

причем массив состоит из десятичных упакованных чисел длиной 5. Пусть места для переменной Z и паспорта P массива a резервированы командами

```
Z     DS    PL5
P     DS    F          для A0
L     DS    H          нижняя граница
U     DS    H          верхаяя граница
      DC    H'5'       длина элемента
```

Предположим, что команды внешнего блока уже заполнили значениями соответствующих границ полуслова L и U. Тогда программа резервирования памяти и вычисления A0 будет такой:

```
      LA        0, 1
      AH        0, U          u + 1
      SH        0, L          u — 1 + 1
      MH        0, P + 8      5 × (u — 1 + 1) — длина
*                               массива
      AH        0, =H'7'      округление до 8
      N         0, =F'—8'     в бо́льшую сторону
      GETMAIN   R, LV=(0)     в регистре 1 — адрес на-
*                               чала участка
      LH        0, L
      MH        0, P + 8      5 × 1 в регистре 0
      SR        1, 0          A0 в регистре 1
      ST        1, P          A0 в слове P
```

Для округления мы воспользовались следующей процедурой: прибавили к числу в регистре 0 число 7, затем «стерли» у результата три младших двоичных разряда. Последнее дей-

ствие производится логической операцией N (см. следующую главу), вторым операндом которой служит двоичный код длинного числа, состоящий из трех нулей в младших позициях и двадцати девяти единиц в старших позициях.

При составлении программы

$$z := a[3 \times (i + j) + 2]$$

предположим, что двоичные значения переменных i и j находятся в регистрах, например, 5 и 7, а общий регистр 1 свободен

```
LA          1, 0(5, 7)          i + j в регистре 1
MH          1, =H'15'           15×(i+j) в регистре 1
A           1, P                A0 + 15×(i + j)
ZAP         Z, 10(5, 1)         z := a[3×(i + j) + 2]
```

Напомним, что адрес элемента a[3 × (i + j) + 2] есть

$$A0 + 15 \times (i + j) + 10.$$

Третья программа, сообщающая операционной системе об освобождении участка памяти, может выглядеть так:

```
* повторить вычисление длины участка в регистре 0
LH          1, L
MH          1, P + 8            5×l в регистре 1
A           1, P                A0+5×l в регистре 1
FREEMAIN    R, LV=(0), A=(1)
```

Мы видим, что процедуры заказа памяти и ее освобождения довольно громоздки. Можно было бы при вычислении длины участка в первой программе опустить команды округления длины (супервизор автоматически округлит заказ). Но при освобождении участка необходимо указать точную длину, и там округление необходимо.

Можно сократить длину программы выхода, добавив к первой программе еще две команды, запоминающие точную длину заказанного участка и адрес его начала в соответственно полуслове HL и слове FA. Тогда третья программа будет содержать только две команды

```
LH          0, HL
FREEMAIN    R, LV=(0), A=FA
```

Нужно также иметь в виду, что сами процедуры GETMAIN и FREEMAIN очень сложны и требуют много машинного времени. Поэтому к ним следует прибегать в программе не слишком часто, заказывая и освобождая память по возможности крупными участками.

Глава 4

ДЕЙСТВИЯ НАД ТЕКСТАМИ И КОДАМИ

§ 4.1. Логические операции

Операндами подавляющего большинства рассмотренных до сих пор машинных команд были числа, представленные в том или ином виде: в двоичной или десятичной системах счисления, целые числа или числа с плавающей точкой. В отличие от этого операнды логических команд машины не имеют какого-либо фиксированного смысла и могут интерпретироваться различным образом при обработке информации произвольной природы: литературных текстов, изображений лунных ландшафтов, данных переписи населения и т. д. Однако, имея в виду особенности алгоритмов конкретных операций, удобно представлять себе их операнды в одних случаях как последовательности байтов, а в других — как последовательности битов. Например, операнды команды переноса MVC естественно рассматривать как последовательности некоторых символов, каждый из которых представлен комбинацией из восьми битов, хранящейся в байте оперативной памяти. В отличие от этого первый операнд команды SLL логического сдвига влево следует рассматривать как последовательность, состоящую из тридцати двух битов. В дальнейшем мы обычно будем называть операнды первого типа текстами, а второго — кодами.

Логические операции могут вызывать прерывание программы по одной из трех причин:

А — превышение максимального адреса в конкретной установке,
З — нарушение защиты памяти при записи результата,
С — нарушение спецификации операнда.

Исключения из этого правила составляют только команды редактирования ED и EDMK, вторые операнды которых — десятичные упакованные числа или последовательности таких чисел. Эти команды могут вызвать прерывание Д по ошибке в коде цифры или знака.

Разные логические команды имеют форматы всех пяти типов: RR, RX, RS, SS и SI. В командах формата SS длина одного или обоих операндов указывается лишь в поле первого операнда и может меняться в пределах от 1 до 256.

В табл. 8 приведены все логические команды машины в явной форме ассемблера. Неявное указание их операндов ничем не отличается от неявного указания операндов рассмотренных ранее операций соответствующих форматов. Рассмотрим табл. 8.

Первые четыре команды являются модификациями операции переноса (MoVe). Команда MVC переносит последовательность символов (Character), начало которой указано адресом $D_2(B_2)$, на место последовательности, начинающейся с адреса, указанного смещением D_1 и базисным регистром B_1 на поле первого операнда. Длина последовательности указана в поле первого операнда числом L, не превышающим 256. Перенос происходит последовательно, порциями в один байт, в порядке возрастания адресов, независимо от того, перекрываются занятые операндами участки в оперативной памяти или нет. Например, команда

```
MVC     B + 1 (25), B
```

копирует код, хранившийся ранее в байте с адресом B, в 25 байтов с адресами B + 1, B + 2, ..., B + 25, стирая то, что было в этих байтах до выполнения команды.

Действие MVN отличается от действия MVC лишь тем, что переносятся только правые, или цифровые (Numerics), половинки байтов, а левые остаются без изменения. Команда MVZ переносит лишь левые, или зонные (Zones), половинки байтов, не изменяя их правых частей. Команда MVI переносит один байт, записанный непосредственно (Immediate) в поле второго операнда, на место, указанное адресом $D_1(B_1)$. Следующие четыре команды представляют модификации операции логического сравнения (Compare Logical). Эти команды вырабатывают в регистре CC код, определяющий результат сравнения операндов. Операнды рассматриваются как последовательности битов, а сравнение производится бит за битом в порядке слева направо. Меньшим считается тот операнд, у которого первый бит, отличный от соответствующего бита другого операнда, имеет значение 0. Коды сравнения 00, 01 и 10 означают соответственно, что первый операнд равен, меньше или больше второго. Операндами модификаций являются: коды в общих регистрах R_1 и R_2, коды в регистре R_1 и слове с адресом $D_2(X_2, B_2)$, коды длиной $8 \times L$ битов, начинающиеся с левых битов байтов с адресами $D_1(B_1)$ и $D_2(B_2)$, восьмибитовые коды байта с адресом $D_1(B_1)$ и непосредственного операнда I_2.

Четыре следующие команды являются модификациями логической операции «И» (aNd). Операндами модификаций служат последовательности битов, по размеру и расположению

Таблица 8

Код операции	Операнды	Прерывания	Код условия				Описание операции	Время в мсек
			00	01	10	11		
1	2	3	4	5	6	7	8	9
MVC	D_1 (L, B_1), D_2 (B_2)	A, З					Перенос байтов	11,38 *
MVN	D_1 (L, B_1), D_2 (B_2)	A, З					Перенос правых полубайтов	10,57 *
MVZ	D_1 (L, B_1), D_2 (B_2)	A, З					Перенос левых полубайтов	12,60 *
MVI	D_1 (B_1), I_2	A, З					Перенос непосредственного операнда	4,87
CLR	R_1, R_2		=	<	>		Сравнение регистров	1,02
CL	R_1, D_2 (X_2, B_2)	A, C	=	<	>		Сравнение регистра со словом	4,47
CLC	D_1 (L, B_1), D_2 (B_2)	A	=	<	>		Сравнение текстов	11,78 *
CLI	D_1 (B_1), I_2	A	=	<	>		Сравнение непосредственное	4,87
NR	R_1, R_2		0	┐0			Операция «и» над регистрами	1,01
N	R_1, D_2 (X_2, B_2)	A, C	0	┐0			Операция «и» между регистром и словом	4,46
NC	D_1 (L, B_1), D_2 (B_2)	A, З	0	┐0			Операция «и» над текстами	12,99 *
NI	D_1 (B_1), I_2	A, З	0	┐0			Операция «и» непосредственная	7,31
OR	R_1, R_2		0	┐0			Операция «или» над регистрами	1,02
O	R_1, D_2 (X_2, B_2)	A, C	0	┐0			Операция «или» между регистром и словом	4,46
OC	D_1 (L, B_1), D_2 (B_2)	A, З	0	┐0			Операция или» над текстами	12,99 *
OI	D_1 (B_1), I_2	A, З	0	┐0			Операция «или» непосредственная	7,31

XR	R_1, R_2		0	┐0			Отрицание равнозначности над регистрами	1,01
X	R_1, D_2 (X_2, B_2)	A, C	0	┐0			Отрицание равнозначности регистра и слова	4,46
XC	D_1 (L, B_1), D_2(B_2)	A, З	0	┐0			Отрицание равнозначности над текстами	12,90 *
XI	D_1 (B_1), I_2	A, З	0	┐0			Отрицание равнозначности непосредственное	7,32
TM	D_1 (B_1), I_2	A	0	0 и 1		1	Проверка по маске	5,49
IC	R_1, D_2 (X_2, B_2)	A					Загрузка байта	4,46
STC	R_1, D_2 (X_2, B_2)	A, З					Выгрузка байта	4,87
SLL	R_1, D_2 (B_2)						Сдвиг логический влево	3,25 **
SRL	R_1, D_2 (B_2)						Сдвиг логический вправо	3,65 **
SLDL	R_1, D_2 (B_2)	C					Двойной логический сдвиг влево	4,47 **
SRDL	R_1, D_2 (B_2)	C					Двойной логический сдвиг вправо	4,47 **
TS	D_1 (B_1)	A, З	Был 0	Была 1			Проверка и запись байта	5,69
TR	D_1 (L, B_1), D_2 (B_2)	A, З					Перекодировка	20,87 *
TRT	D_1 (L, B_1), D_2 (B_2)	A	****	*****	******		Поиск специального символа	19,06 *
ED	D_1 (L, B_1), D_2 (B_2)	A, З,					Редактирование чисел	459,17 ***
EDMK	D_1 (L, B_1), D_2 (B_2)	A, З,					Редактирование с отметкой	460,56 ***

* Время указано для операнда в 4 байта.
** Время указано для сдвига на 3 бита.
*** Время указано для операнда в 120 байтов.
**** Разделитель не обнаружен.
***** Разделитель внутри текста.
****** Разделитель в конце текста.

аналогичные операндам соответствующих модификаций логического сравнения. Операция производится отдельно над каждой парой соответствующих битов операндов и ее результат замещает соответствующий бит первого операнда. Он вырабатывается по известному правилу (табл. 9).

Четыре следующие команды — модификации логической операции «ИЛИ» (Or) Они отличаются от модификаций «И» лишь правилом вычисления результата (табл. 10).

Таблица 9

Операнд 1	Операнд 2	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Таблица 10

Операнд 1	Операнд 2	Результат
0	0	0
0	1	1
1	0	1
1	1	1

Таблица 11

Операнд 1	Операнд 2	Результат
0	0	0
0	1	1
1	0	1
1	1	0

И, наконец, четыре следующие команды составляют модификации логической операции «исключающие или» (eXclusive or) (табл. 11), обратной логической операции «тождество».

Команда TM проверки по маске (Test under Mask) вырабатывает код условия, определяющий соответствие первого операнда — восьмибитового кода в байте с адресом $D_1(B_1)$ и непосредственного операнда I_2, играющего роль маски. Рассматриваются только те биты первого операнда, номера которых (из диапазона 0—7) совпадают с номерами тех битов маски, которые имеют единичные значения. Если все выделенные таким образом биты первого операнда имеют значение 0, вырабатывается код условия 00. Если все выделенные биты имеют значение 1, вырабатывается код условия 11. Во всех остальных случаях (т. е. при наличии как нулей, так и единиц) вырабатывается код 01.

Операция IC загрузки байта (Insert Character) загружает код, хранящийся в байте с адресом $D_2(X_2, B_2)$, в биты с номерами 24—31 общего регистра R_1, не меняя значений остальных битов регистра. Обратная операция STC выгрузки байта (STore Character) копирует содержимое последних восьми битов регистра R_1 в байт с адресом $D_2(X_2, B_2)$.

Следующие четыре команды осуществляют модификации логического сдвига (Shift Logical). Вторая буква в коде операции указывает направление сдвига: влево (Left) или вправо (Right). Буква D означает операцию двойного (Double) сдвига. Запись на поле второго операнда в этой команде указывает не адрес в памяти, а непосредственно величину сдвига в битах. Точнее, величина сдвига определяется числом, представленным младшими шестью битами двоичного представления

суммы смещения D_2 и числа в базисном регистре B_2. Операндом простого сдвига является код в общем регистре R_1. Операция заключается в сдвиге этого кода вправо или влево на указанное число битов. При сдвиге вправо на освободившиеся старшие позиции регистра заносятся нули, а биты, выдвигающиеся из регистра вправо, пропадают. При сдвиге влево пропадают старшие биты кода, а справа дописываются нули.

Операндом двойного сдвига служит код из 64 битов, расположенных в двух общих регистрах с номерами R_1, $R_1 + 1$, содержащих соответственно левую и правую половины кода. При этом номер R_1 обязан быть четным. При нарушении этого условия происходит прерывание по неправильной спецификации операнда. При двойном сдвиге регистры R_1 и $R_1 + 1$ соединяются в один регистр из 64 позиций. В остальном операция двойного сдвига аналогична простому сдвигу.

Команда TS «проверить и записать» (Test and Set) записывает в байт с адресом $D_2(B_2)$ код из восьми единиц и вырабатывает в CC код, характеризующий прежнее значение в $D_2(B_2)$. Именно код 00 вырабатывается в том случае, когда до выполнения команды в самом левом бите байта $D_2(B_2)$ хранился ноль. В противном случае вырабатывается код 01. Проверка значения и записи производится в одном машинном такте, составляя единое действие. Эта операция предназначена для синхронизации процессов при мультипрограммировании.

О четырех оставшихся операциях мы пока говорить не будем, отложив знакомство с ними до конца главы.

В дальнейшем мы будем пользоваться командой ассемблера DC для заказа логических постоянных — последовательностей символов или битов. Формат ее таков:

[имя] DC [кратность] тип [модификатор длины] 'величина'.

Тип задается одной из букв: C, X или B. Тип C означает, что постоянная задается произвольной последовательностью символов из алфавита машины. Величина постоянных типов X или B задается соответственно последовательностью 16-ричных или двоичных цифр. Модификатор длины состоит из буквы L и следующего за ней десятичного целого числа, указывающего длину постоянной в байтах. Если модификатора нет, для постоянной отводится минимальное необходимое количество байтов, причем постоянные типов X и B дополняются слева, если это необходимо, несколькими нулевыми битами до целого количества байтов. Если указан модификатор длины, постоянная типа C справа дополняется пробелами или усекается до нужной длины. В случае постоянных типов X или B усечение или дополнение нулевыми битами производится слева. Подробнее об этом см. в [5].

Мы уже встречались ранее в нескольких примерах (см. примеры 3—6 гл. 3) с необходимостью использования логических

операций. Познакомимся теперь с типичными классами задач, требующих широкого использования этих операций, и некоторыми стандартными приемами их решения.

§ 4.2. Действия с логическими значениями

Во многих задачах автоматической обработки информации приходится пользоваться логическими переменными (в смысле языка АЛГОЛ-60) и вычислять значения составленных из них логических выражений. Иногда эти значения удобно хранить в оперативной памяти, а иногда представлять в виде кода условия в регистре СС. Рассмотрим сначала основные операции с логическими переменными, представленными в оперативной памяти кодами полных байтов.

Пусть X и Y — ассемблерные имена байтов, хранящих значения логических переменных x и y. Значение **true** условимся представлять кодом X'FF', а значение **false** — кодом X'00'. Тогда основные логические операции можно осуществить следующими машинными командами:

```
XI      X, 255      x := ¬x
NC      X, Y        x := x ∧ y
OC      X, Y        x := x ∨ y
```

Мы видим, что каждая операция производится одной машинной командой. Операции же импликации и тождества требуют по две команды. Например, оператор присваивания

$$x := x \supset y,$$

если его заменить эквивалентным оператором

$$x := \neg x \vee y,$$

можно выполнить так:

```
XI      X, 255      x := ¬x
OC      X, Y        x := x ∨ y
```

Оператор

$$x := x \equiv y,$$

преобразовав его к эквивалентной форме

$$x := \neg(\neg(x \equiv y)),$$

можно осуществить двумя командами:

```
XC      X, Y        x := ¬(x ≡ y)
XI      X, 255      x := ¬x
```

Рассмотренные приемы могут применяться при вычислении произвольного логического выражения, составленного только

из логических переменных. Предварительно это выражение следует преобразовать к эквивалентному, вычисление которого сводится к цепочке операций $\neg$, $\wedge$ и $\vee$.

Пример 1. Написать программу оператора присваивания

$$p := x \vee y \supset z \wedge t$$

Предполагается, что все логические переменные представлены байтами, указанными ассемблерными именами P, X, Y, Z, T, имеющими характеристики длины 1.

Преобразуем наш оператор к виду

$$p := \neg (x \vee y) \vee (z \wedge t)$$

Нам понадобится рабочая переменная, которую мы можем заказать командой ассемблера

```
W       DC      C
```

Программу можно написать так:

```
MVC     P, X        p := x
OC      P, Y        p := x ∨ y
XI      P, 255      p := ¬(x ∨ y)
MVC     W, Z        w := z
NC      W, T        w := z ∧ t
OC      P, W        p := p ∨ w
```

Значение логического выражения, входящего в состав условия, целесообразно вычислять в виде кода условия. Это сделать очень просто, если логическое выражение есть отношение (см. гл. 2). Если же вычисление выражения сводится к последовательности операций $\neg$, $\wedge$, $\vee$, можно воспользоваться кодом условия, который вырабатывает последняя операция. Например, если логическое выражение в условии совпадает с правой частью оператора в предыдущем примере, то, воспользовавшись рабочими переменными p и w, можно вычислить нужное значение той же последовательностью из шести команд, которая составляет решение примера 1. В результате выполнения последней команды

```
OC      P, W
```

в регистре CC появится код 00, если наше выражение имеет значение **false**, и код 01 в противном случае.

Иногда требуется «извлечь» логическое значение из регистра CC и записать его в память в виде кода байта, имеющего значение **true** или **false**.

Пример 2. Написать программу оператора присваивания

$$p := x \vee a \leqslant b.$$

Здесь p и x — логические переменные, хранящиеся в байтах P и X, a и b — короткие двоичные числа в полусловах A и B.

```
LH      0, A      a в регистре 0
CH      0, B      CC := 10 в случае false
MVI     P, 0      p := false
BH      *+8       переход при false в CC
MVI     P, 255    p := true
OC      P, X      p := p ∨ x
```

Первая и вторая команды вычисляют значение отношения в CC. Третья, четвертая, пятая команды пересылают значение отношения $a \leqslant b$ из регистра CC в байт P. Напоминаем, что команда MVI не меняет кода условия и команда BH воспринимает код условия, выработанный командой CH.

Обратная операция — занесение в регистр CC логического значения, представленного байтом X, — осуществляется одной командой

```
CLI     X, 0      CC := if x then 10 else 00
```

Если логическое выражение составляет условие, часто оказывается выгодным вместо прямого вычисления в регистре CC значения выражения преобразовать исходный условный оператор к цепочке условных операторов с более простыми условиями.

Пример 3. Написать программу оператора

$$\textbf{if } a < b \vee c \leqslant d \textbf{ then go to } N1 \textbf{ else go to } N2$$

Предполагается, что a, b, c и d — числа типа E, расположенные в словах A, B, C и D, а N1 и N2 — ассемблерные имена ветвей программы.

Заменим исходный оператор тремя операторами:

$$\textbf{if } a < b \textbf{ then go to } N1;\quad \textbf{if } c \leqslant d \textbf{ then go to } N1;\quad \textbf{go to } N2$$

Теперь машинную программу написать совсем просто:

```
LE      0, A      a в регистре 0 с плавающей точкой
CE      0, B      if a < b then
BL      N1        go to N1
LE      0, C      c в регистре 0 с плавающей точкой
CE      0, D      if c ⩽ d then
BNH     N1        go to N1
B       N2        go to N2
```

Если имеется достаточно большой массив логических значений, для их хранения можно воспользоваться более компактной формой, изображая каждое значение одним битом. Тогда каждый байт может содержать восемь значений. Рассмотрим приемы, которые позволяют, зная адрес байта и номер бита, а) извлечь значение, переведя его в рассмотренную выше форму, б) записать логическое значение в нужный бит, не меняя значений остальных семи.

Пример 4. Требуется а) извлечь из байта X логическое значение x_5, изображаемое его пятым битом, записав его в виде байта с адресом A, б) записать в первый бит байта X логическое значение, заданное целым байтом B. Напомним, что биты в байте нумеруются в порядке слева направо цифрами от 0 до 7.

Задачу а) можно решить четырьмя командами:

```
TM    X, B'100'     CC := if x5 then 11 else 00
MVI   A, 0          A := false
BZ    *+8           переход при false в CC
MVI   A, 255        A := true
```

Если бы мы хотели представить значение x_5 в регистре CC, достаточно выполнить лишь первую команду.

Для решения задачи б) в общем случае также нужны четыре команды:

```
NI    X, B'10111111'   x1 := false
CLI   B, 0             CC := if b then 10 else 00
BE    *+8              переход при false в B
OI    X, B'10000000'   x1 := true
```

Первая команда записывает 0 в первый бит байта X, не меняя значения остальных битов. Вторая переносит значение B в регистр CC. Третья команда производит обход четвертой, если значение B есть **false**. Четвертая команда заносит 1 в первый бит B, не меняя значения остальных битов.

Задачу б) можно было бы решить и «обратным» способом, занеся первой командой

```
OI    X, B'1000000'
```

в первый бит X значение **true**, а последней командой

```
NI    X, B'10111111'
```

заменить его, если нужно, на **false**. Конечно, в этом варианте нужно заменить и третью команду на

```
BNZ   *+8
```

§ 4.3. Действия с кодами

Как уже не раз упоминалось выше, мы называем кодами информацию неизвестной природы, представленную в машине последовательностью «равноправных» битов. Практика программирования выявила ряд типичных операций над кодами, к которым приходится прибегать в достаточно широких классах случаев. Набор логических команд каждой универсальной машины позволяет выполнить тем или иным путем любую из этих операций. Познакомимся с такими операциями и способами их реализации в машинах серии ЕС ЭВМ. Для простоты будем считать, что все рассматриваемые коды содержат не более 32 битов и расположены в полных словах или общих регистрах.

Пример 5. Выделение части кода по маске.

Пусть в слове F1 хранится некоторый код, а в слове M — код, играющий роль маски: единицы в нем указывают те позиции, которые нужно перенести из кода F1 в слово F2, записав во все остальные биты слова F2 нули.

Решение по существу сводится к машинной команде «И»:

```
L     0, F1     копирование в общий регистр
N     0, M      стирание «лишних» битов F1
ST    0, F2     перенос результата в слово F2
```

Пример 6. Инвертирование кода.

Пусть в слове F1 находится некоторый код. Нужно в слове F2 образовать код, все биты которого имеют значения, обратные значениям соответствующих битов кода F1.

Эта задача также сводится к машинной команде «исключающее или», вторым операндом которой служит код из 32 единиц. Он совпадает с представлением двоичного длинного числа —1 в дополнительном коде.

```
L     0, F1          копирование F1 в общий регистр 0
X     0, =F'—1'      инвертирование в регистре 0
ST    0, F2          запись результата в F2
```

Пример 7. Слияние двух кодов.

В словах F1 и F2 хранятся два кода. Нужно в слове F2 образовать новый код, который состоит частично из битов кода F1, частично из битов кода F2. Позиции выделяемых битов в общем случае указываются двумя «непересекающимися» масками M1 и M2, причем общее количество единиц в M1 и M2 может быть меньше 32.

Рассмотрим конкретный пример, когда M1 состоит из единиц на всех нечетных позициях слова, а M2 — из единиц на

всех четных позициях. Такие маски можно заказать следующими командами ассемблера:

```
        DS      0F          выравнивание на границу слова
M1      DC      4X'55'      единицы в нечетных позициях
M2      DC      4X'AA'      единицы в четных позициях
```

Программа может выглядеть так:

```
        L       0, F1       копирование F1 в общий регистр 0
        N       0, M1       выделение нечетных разрядов F1
        L       1, F2       копирование F2 в общий регистр 1
        N       1, M2       выделение четных разрядов F2
        OR      0, 1        слияние частей в регистре 0
        ST      0, F3       запись результата в F3
```

Пример 8. Подсчет количества единиц в коде.

Пусть в слове F хранится некоторый код. Нужно в полуслове H в виде короткого двоичного числа указать количество единиц в коде F.

Задачу можно решить, например, так: поместив код в нечетный регистр, сдвигать его последовательно на один бит операцией двойного сдвига влево в соседний регистр, проверяя каждый раз наличие единицы в разряде 31 этого регистра.

```
        L       1, F        загрузка кода в регистр 1
        LA      14, 1       загрузка 1 в разряд 31 ре-
*                           гистра 14
        SR      15, 15      очистка регистра для под-
*                           счета единиц
        LA      2, 32       счетчик цикла
LOOP    SLDL    0, 1        выдвижение крайнего ле-
*                           вого разряда из регистра 1
*                           в регистр 0
        NR      0, 14       CC := 01 при наличии еди-
*                           ницы
        BZ      *+8         обход следующей команды
        AR      15, 14      подсчет единиц
        BCT     2, LOOP
        STH     15, H       запись результата в H
```

Предварительная очистка регистра 0 не требуется, так как при выполнении команды

```
        NR      0, 14
```

все разряды общего регистра 0, кроме, может быть, разряда 31, заполняются нулями.

Пример 9. Сборка кода по маске.

Пусть в слове F хранится некоторый код, а в слове M — другой, играющий роль маски. Необходимо собрать в правой части кода R вплотную к друг другу все те биты кода F, которые стоят в позициях, указанных единицами маски, а левую часть R дополнить нулями. Например, если мы ограничимся кодами длины 8, а значения F и M соответственно: B′10110101′ и B′11001110′, то значением R будет код B′00010010′, т. е. биты F с номерами 0, 1, 4, 5, 6 займут в R позиции соответственно 3, 4, 5, 6, 7.

Для решения задачи удобно воспользоваться специальными кодами — указателями позиции в R и в M, которые мы назовем соответственно RP и MP. Оба кода имеют нули во всех позициях, кроме одной, где стоит единица. Перед началом работы единицы в обоих кодах должны располагаться в позициях 31.

Идея алгоритма заключается в следующем. Мы постепенно сдвигаем влево указатель маски MP. При каждом его совпадении с единицей маски мы проверяем наличие единицы в соответствующей позиции кода F. Если единица обнаружена, мы с помощью операции «ИЛИ» вставляем единицу в очередную позицию регистра R, который в начальный момент заполнен нулями. Очередная позиция R указывается кодом RP, который сдвигается на один разряд влево при каждом совпадении MP с единицей маски M.

Напишем сначала этот алгоритм на алголоподобном языке

```
MP := RP := 1;   R := 0;
                for i := 1 step 1 until 32 do
begin if MP ∧ M ≠ 0 then begin if MP ∧ F ≠ 0 then
      R := R ∨ RP;  L1 : RP ← end;
      L2 : MP ←
end
```

Отличие от эталонного языка состоит в том, что мы ввели в рассмотрение переменные MP, RP, R и P не предусмотренных в языке типов и пользуемся новыми операциями: ← (сдвиг влево), ∧ («И»), ∨ («ИЛИ»), точный смысл которых определится лишь после перевода на машинный язык. Подобные вольности будут допускаться и в дальнейшем без особых оговорок. Дело в том, что АЛГОЛ-60 не приспособлен для описания действий над текстами и кодами, а изучение необходимых элементов какого-либо точного алгоритмического языка с более широкими возможностями (например, АЛГОЛ-68, ПАСКАЛЬ или PL-1) увело бы нас далеко в сторону.

Предположим, что результат R нужно получить в общем регистре 2. Кроме того, нам понадобятся рабочие регистры,

Пусть это будут: 0 и 1 — для переменных MP и RP, 14 — для вычисления левых частей отношений, 15 — для счетчика цикла

```
        LA      0, 1        MP := 1
        LR      1, 0        RP := 1
        SR      2, 2        R := 0
        LA      15, 32      счетчик цикла
LOOP    L       14, M       маска в регистре 14
        NR      14, 0       MP ∧ M в регистре 14
        BE      L2          CC есть 00 при MP ∧ M = 0
        N       14, F       MP ∧ F в регистре 14
        BE      L1          CC есть 00 при MP ∧ F = 0
        OR      2, 1        R := R ∨ RP
L1      SLL     1, 1        сдвиг RP влево на один бит
L2      SLL     0, 1        сдвиг MP влево на один бит
        BCT     15, LOOP    проверка конца цикла
```

В этой программе в пояснении нуждается лишь, может быть, команда

```
        N       14, F
```

Значение ее регистрового операнда перед исполнением совпадает с MP, так как предыдущая команда передает ей управление лишь в том случае, когда предшественница этой последней — команда NR — образовала в регистре 14 не нулевой результат.

В некоторых машинах операция сборки по маске реализуется специальной машинной командой.

Рекомендуем читателю в качестве упражнения написать программу обратной задачи: по результату R, маске M и значению F с нулями против единиц маски восстановить исходное значение F.

§ 4.4. Действия с текстами

В этом разделе на ряде примеров мы рассмотрим употребление логических операций для обработки информации, представленной в памяти машины текстами — произвольными последовательностями байтов. Кроме того, будут рассмотрены логические операции TR, TRT, ED, EDMK, команда перехода EX и команда ассемблера ORG.

Пример 10. Перенос текста на новое место.

Предположим, что текст длиной l, начинающийся с байта, адрес которого обозначен ассемблерным именем A, нужно перенести на новое место, начинающееся с байта, указанного именем B. Длина текста известна в момент составления программы и буква l замещает соответствующее десятичное число.

Если участки памяти длины l, начинающиеся с байтов A и B, не перекрываются, а l не превышает 256, очевидно, что задача решается одной командой:

```
MVC     B(l), A
```

Это же решение верно и в том случае, когда участки пересекаются так, что байт B расположен левее байта A, т. е. адрес B меньше адреса A (рис. 20). Действительно, операция MVC состоит в последовательном переносе информации порциями в один байт с поля, занимаемого вторым операндом, на поле первого в порядке возрастания адресов.

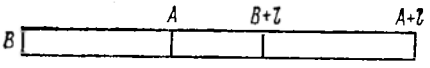


Рис. 20. Сдвиг текста влево

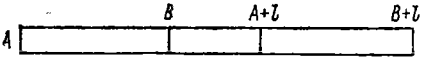


Рис. 21. Сдвиг текста вправо

Если же поля пересекаются так, что байт B расположен правее байта A (рис. 21), то очевидно, что одной командой осуществить перенос нельзя: часть текста, находившаяся до исполнения команды MVC на участке от B до A + l будет испорчена.

Предположим, что имеется свободный участок C длиной не меньше чем l. Тогда перенос можно осуществить через этот участок двумя командами:

```
MVC     C(l), A
MVC     B(l), C
```

Более экономный с точки зрения расхода памяти способ состоит в использовании свободного участка с длиной, равной длине перекрывающейся части полей A и B.

```
        MVC     C(A+l-B), B     перенос конца A
*                               на свободное
*                               место
        MVC     B(B-A), A       перенос начала
*                               текста A
        MVC     B+B-A(A+l-B), C перенос конца
*                               текста A
```

Если длина A + l — B общей части полей не превосходит длины оставшихся частей, можно осуществить перенос в две команды, не занимая дополнительной памяти:

```
        MVC     B+B-A(A+l-B), B перенос общей
*                               части в конец
*                               поля B
        MVC     B(B-A), A       перенос нача-
*                               ла A в начало B
```

Заметим, что B + B — A является переместимым выражением, а A + 1 — B есть абсолютное выражение (см. [5, раздел 2.1.7]).

Можно также воспользоваться циклической программой, переносящей текст с поля A на B побайтно, начиная с правых байтов:

```
         LH     1, =H'1'      счетчик цикла в
*                             регистре 1
LOOP     IC     0, A-1(1)     очередной байт
*                             в регистре 0
         STC    0, B-1(1)     перенос байта
*                             на поле B
         BCT    1, LOOP
```

Эта программа справедлива при любом $1 \leqslant 2^{15} - 1$.

Пример 11. Перенос текста переменной длины.

Предположим, что поля A и B длиной $l \leqslant 256$ перекрываются так, что перенос с поля A на B возможен одной командой MVC. Длина переносимого текста l не известна в момент написания программы, а вычисляется во время ее исполнения в общем регистре 5 в виде двоичного числа.

Задачу можно решить следующим путем:

```
         STC    5, M+1        запись в поле
*                             длины следую-
*                             щей команды
M        MVC    B(1), A       перенос текста
```

В регистре 5 должна находиться «машинная» длина текста, которая на 1 меньше настоящей длины. Команда STC вписывает эту длину во второй байт команды MVC. Поэтому команда MVC, исполняющаяся после завершения STC, перенесет с поля A на поле B текст нужной длины.

Решение плохо тем, что программа, содержащая такой фрагмент, не может быть повторно входимой * в силу того, что содержит изменяющиеся в процессе исполнения команды. Лучшее решение состоит в употреблении команды перехода «выполнить», имеющей символическое имя EX (EXecute). Команда имеет формат RX и в явной форме ассемблера имеет вид

EX R_1, $D_2(X_2, B_2)$

Выполнение команды EX сводится к исполнению команды, хранящейся по адресу $D_2(X_2, B_2)$, с предварительным изменением второго байта этой команды посредством последних восьми битов общего регистра R_1. Если команда, хранящаяся по адресу $D_2(X_2, B_2)$, не есть команда перехода, преемницей команды EX будет непосредственно расположенная вслед за EX команда.

* Свойство повторной входимости программы (или реентерабельность программы) подробно рассматривается в гл. 5 (см. также [13, с. 97]).

Изменение второго байта команды, хранящейся по адресу $D_2(X_2, B_2)$, состоит в наложении на него последнего байта из регистра R_1 в смысле операции «или».

На поле C в области постоянных программы, начинающемся на границе полуслова, расположим команду

```
C       MVC     B(1), A
```

а в том месте программы, где необходима пересылка, напишем команду

```
        EX      5, C
```

В этом варианте в байте C + 1 (втором байте команды MVC) должен храниться нулевой код, * в то время как значение этого байта в предыдущем варианте безразлично.

Командой EX можно воспользоваться для изменения второго байта команды любого формата: SS (длина одного или двух операндов), SI (непосредственный операнд), RS (R_1 и R_3), RX (R_1 и X_2).

Пример 12. Поиск по ключу в неупорядоченной таблице.

Пусть текст, занимающий участок памяти от байта A до байта B—1 включительно, состоит из последовательности отрезков в l_2 байтов каждый, причем в каждом из них первые l_1 байтов составляют идентифицирующую информацию. Нужно найти адрес начала первого слева участка, идентифицирующая информация которого (называемая обычно ключом) совпадает с текстом, расположенным на поле W. В случае успешного завершения поиска нужно передать управление по адресу YES, записав предварительно найденный адрес в регистр 1, а в случае неудачи — по адресу NO.

Решение задачи в предположении, что l_1 и l_2 известны при написании программы и $l_1 \leqslant 256$, может быть таким:

```
        LA      1, A            адрес начала поиска в
*                               регистре 1
        LH      14, =H'l2'      шаг поиска в регистре 14
        LA      15, B-l2        конечный адрес в ре-
*                               гистре 15
LOOP    CLC     W(l1), 0(1)     сравнение ключей
        BE      YES             выход из цикла при сов-
*                               падении
        BXLE    1, 14, LOOP     продвижение адреса в
*                               регистре 1 и проверка
*                               конца
        B       NO              совпадение не обнару-
*                               жено
```

* Допустим и такой код, который не имеет единиц в битах, отвечающих нулям в регистре R_1.

Пример 13. Перекодировка текста.

Задача перекодировки состоит в замене внутренних кодов символов текста другими комбинациями битов. Соответствие между старым и новым кодами устанавливается таблицей перекодировки, состоящей из 256 упорядоченных пар. Если мы упорядочим элементы таблицы по компонентам, содержащим старые коды, для хранения таблицы в памяти достаточно поле в 256 байтов: старый код символа указывается номером элемента. Обозначив именем TAB адрес первого байта таблицы, мы в байте TAB + i, $0 \leqslant i \leqslant 255$, найдем новый код символа, старый код которого совпадает с двоичным представлением числа i.

Если таблица составлена указанным образом, перекодировка текста длиной $l \leqslant 256$, начинающегося в байте с адресом A, производится одной командой:

```
TR      A(l), TAB
```

которая последовательно, в порядке слева направо, заменяет коды символов текста по правилу, заданному таблицей TAB. Если операнды TR рассматривать как векторы с описанием

integer array A [1 : l], TAB [0 : 255],

то ее действие эквивалентно оператору цикла

for i := 1 **step** 1 **until** l **do** A [i] := TAB [A [i]]

Таблица перекодировки может быть либо постоянной программы, либо построена уже во время ее исполнения.

Рассмотрим конкретную задачу: в тексте, представленном во внутреннем коде ЕС ЭВМ, нужно заменить все квадратные скобки на соответствующие круглые. Длина текста 500 байтов, адрес первого байта указан именем A, таблицу перекодировки можно строить на поле TAB.

```
         LA      1, 255              счетчик цикла
LOOP     STC     1, TAB(1)           построение таблицы, которая
*                                    не меняет текст
         BCT     1, LOOP
         STC     1, TAB              запись X'00' в
*                                    байт TAB
         MVI     TAB + C']', C')'    замена двух элементов
*                                    таблицы
         MVI     TAB + C'[', C'('
         TR      A(250), TAB         замена прямоугольных скобок
*                                    на круглые
         TR      A + 250(250), TAB
```

Первые четыре команды строят таблицу, которая производит тождественное преобразование любого текста. Две следующие команды заменяют два элемента таблицы, соответствующие прямоугольным скобкам, на коды круглых скобок.

Пример 14. Машинная команда TRT и команда ассемблера ORG.

Машинная команда TRT (TRanslate and Test) предназначена для поиска в тексте, составляющем первый операнд, самого левого вхождения одного из символов, указанных специальным образом во втором операнде, длина которого всегда составляет 256.

Если операнды команды

```
TRT     A(l), B
```

рассматривать как векторы с описанием

integer array A[1 : l], B[0 : 255]

то ее действие эквивалентно действиям следующих операторов

```
        for i := 1 step 1 until l do if B[A[i]] ≠ 0 then
begin R1 := адрес A[i]; R2 := B[A[i]]; CC := if i < l then 01
        else 10; go to OUT
end;
        CC := 00;
        OUT: ...
```

Назовем разделителями те символы A[i] текста A, которым отвечают ненулевые коды в таблице B. Тогда можно сказать, что команда TRT находит в тексте длиной не более 256 первый разделитель. Адрес найденного разделителя заносится в регистр 1, код разделителя — в последний байт регистра 2, а результат поиска (разделитель не обнаружен, найден внутри текста, является последним символом) указывается в коде условия CC.

Рассмотрим конкретную задачу. Нужно написать фрагмент программы, который, получив в общем регистре 3 адрес начала поиска, указывает в регистре 1 адрес ближайшего разделителя, а в регистре 2 — его код. При неудаче нужно осуществить переход по адресу NO. Разделителями считаются: запятая, точка с запятой и пробел.

Решение, как в предыдущем примере, сводится к построению таблицы разделителей, которую мы теперь построим статически, т. е. во время компиляции программы. Воспользуемся для этого командой ассемблера ORG, имеющей формат:

```
[имя]     ORG     [переместимое выражение]
```

В области постоянных программы напишем следующую последовательность команд ассемблера:

```
B         DC      256X'00'    заготовка 256 пустых байтов
          ORG     B + C','    запись
          DC      C','        кода запятой
          ORG     B + C';'    запись
```

```
DC      C';'        кода точки с запятой
ORG     B + C' '    запись
DC      C' '        кода пробела
ORG     ,           возвращение к счетчику
                    B + 256 *
```

При распределении памяти для машинных команд, постоянных и переменных, ассемблер использует свою переменную, называемую счетчиком адреса. Первому байту программы сопоставляется значение счетчика адреса, равное нулю. После обработки каждой команды входной программы значение счетчика увеличивается на длину соответствующего элемента машинной программы, указывая на следующий свободный байт. Команда ORG представляет исключение из правила. В результате ее обработки счетчик адреса приобретает значение, отвечающее той точке программы, на которую указывает перемещаемое выражение в поле операндов. Эта точка может лежать как ниже, так и выше текущего места. Необходимо лишь, чтобы значение всех имен, входящих в состав выражения, было определено на поле имени в ранее обработанных командах. Если же выражение отсутствует, счетчик адреса приобретает максимальное из всех значений, которые он приобретал ранее.

В нашем случае счетчик адреса изменяется следующим образом.

После обработки команды

```
B       DC      256X'00'
```

он приобретает значение, соответствующее адресу B + 256. В результате обработки

```
        ORG     B + C','
```

он уменьшится до значения адреса B + C','. После обработки команды

```
        DC      C','
```

он увеличит свое значение на единицу. После следующей команды он переместится в точку B + C';', и т. д. После обработки команды

```
        ORG
```

он вернется к значению B + 256.

---

* Если команда ассемблера или макрокоманда (например, ORG, END, EXIT и пр.) допускает как наличие, так и отсутствие операндов, а мы хотим снабдить ее комментарием, необходимо указать отсутствие операндов запятой со следующим за ней по крайней мере одним пробелом. В противном случае компилятор ассемблера не сможет отличить комментарий от операндов.

В области команд программы следует написать две машинные команды:

```
        TRT     0(256, 3), B    поиск разделителя
        BE      NO              переход при отсутствии разде-
*                               лителя среди 256 литер
```

Пример 15. Действия со списками.

Рассмотрим список простейшей структуры, изображенный на рис. 22. Идентификатором BEG обозначен адрес машинного слова, хранящего адрес указателя первого элемента списка. Каждый элемент состоит из машинного слова — указателя элемента — и следующей непосредственно за указателем информационной части, структура которой для нас сейчас безразлична.

Рис. 22. Список с одной связью

Указатель элемента содержит адрес указателя следующего элемента списка. Указатель последнего элемента содержит нулевой адрес. Список может быть пустым. В этом случае слово BEG содержит нулевой адрес *.

Рассмотрим две простейшие задачи: добавление нового элемента в конец списка и исключение из списка некоторого элемента.

Пусть в слове LAST хранится адрес некоторого элемента, который нужно добавить в конец списка, изображенного на рис. 22. Предположим, что в указатель этого элемента уже записан нулевой адрес.

Решение состоит из двух частей: поиска указателя последнего элемента и замены нуля в этом указателе на адрес из слова LAST.

```
*  Образование в общем регистре 1 адреса указателя последнего
*                                                  элемента
        SR      0, 0            число 0 для поиска нулевого
*                               указателя
        LA      1, BEG          адрес указателя начала в ре-
*                               гистре 1
        B       COMP            к проверке, не пуст ли список?
LOOP    L       1, 0(1)         адрес следующего указателя
*                               в регистре 1
COMP    C       0, 0(1)         проверка указателя на 0
        BNE     LOOP
```

* Подробнее о списках см. [14, разделы 1.4.6 и 1.4.7].

```
* добавление нового элемента
         L      0, LAST     замена указателя, адрес ко-
*                           торого
         ST     0, 0(1)     находится в регистре 1
```

Если список был пуст, то содержимое слова LAST переписывается в слово BEG.

Пусть теперь нужно из списка, изображенного на рис. 22, исключить некоторый (не обязательно последний) элемент. Предположим, что адрес указателя элемента, предшествующего исключаемому, хранится в слове OUT. Например, если мы хотим исключить элемент P2, в слове OUT должен быть адрес P1, а решение сведется к переносу содержимого указателя P2 в указатель P1.

```
         L      1, OUT      в регистре 1 адрес предше-
*                           ствующего указателя
         L      2, 0(1)     в регистре 2 адрес указателя
*                           исключаемого элемента
         L      0, 0(2)     перенос указателя
         ST     0, 0(1)     исключаемого элемента
```

Пример 16. Редактирование чисел по шаблону.
Команда редактирования по шаблону

```
         ED     P(1), N
```

предназначена для переработки десятичного упакованного числа в форму, пригодную для печати. Переработка состоит в замене двоичного кода каждой цифры восьмибитовым кодом соответствующей литеры и в других действиях, определяемых шаблоном P — первым операндом команды. Шаблон может содержать любые символы, в том числе три специальные: X'20' — символ выбора цифры, X'21' — символ начала значимости, X'22' — символ конца поля. Действие команды состоит в записи переработанного числа N на место, занимаемое ранее шаблоном P. Приведем несколько примеров, поясняющих действие команды ED в случае, когда второй операнд есть десятичное упакованное длины три. В этих примерах знак * означает пробел, буква Ц — символ выбора цифры, буква З — символ начала значимости.

шаблон	число	результат
*ЦЦЦЦЦ —	00 00 5+	*****5*
*ЗЦЦЦЦ —	00 00 5+	**0005*
*ЗЦЦЦЦ —	00 00 5—	**0005—
0ЗЦ.ЦЦЦ —	00 00 5—	000.005—
0ЦЦ.ЦЦЦ —	00 00 5+	00000050

Для точного описания алгоритма переработки воспользуемся следующими обозначениями:

ST — триггер знака (одноразрядный регистр процессора), могущий принимать значения 0 или 1;

P [i], i = 1, 2, ..., — позиции шаблона;

N [k], k = 1, 2, 3, ..., — четырехбитовые коды цифры или знака во втором операнде N;

$\overline{N[k]}$ — восьмибитовый код соответствующей цифры второго операнда;

F и R — рабочие переменные, принимающие значения символов шаблона.

В этих обозначениях действие команды ED можно описать так:

```
k := 0; ST := 0; с сброс триггера знака;
F := P [1]; с запоминание первого символа шаблона;
        for i := 1 step 1 until 1 do
begin R := P [i]; с выбор очередного символа шаблона;
случай Ц или З: if R = X'20' ∨ R = X'21' then
    begin k := k + 1;
    P [i] := if ST = 0 ∧ N [k] = 0 then F else N [k]; с изменение
                                                        шаблона;
    if R = X'21' ∨ N [k] ≠ 0 then ST := 1; с изменение триггера
                                                        знака;
    if k — нечетное ∧ N [k + 1] = код знака then
        begin k := k + 1; if N [k] = код плюса then ST := 0 end
        сброс ST по коду плюса
    end случая Ц или З                          else
случай конца поля: if R = X'22' then begin P [i] := F; ST := 0 end
прочие символы: else P [i] := if ST = 0 then F else R
end
```

В нормальном случае общее количество символов Ц и З в шаблоне должно быть равно количеству цифр в операнде N. Если на месте цифры в N окажется неправильный код, возникает прерывание по неправильным данным.

Операнд N может содержать несколько упакованных чисел. В таком случае для сброса триггера знака перед обработкой нового числа нужно отделять части шаблона, относящиеся к одному числу, символами конца поля.

Код условия вырабатывается по следующему правилу:

00 — последнее обработанное число есть ноль или шаблон закончился символом конца поля;
01 — последнее обработанное число отрицательно;
10 — последнее обработанное число положительно.

Команда EDMK производит все действия, предусмотреные в команде ED. Кроме того, она заносит в биты 8—31 общего регистра 1 адрес каждого байта шаблона, в который записывается код ненулевой цифры при триггере знака в состоянии 0. Это дополнительное действие можно использовать для отметки первой значащей цифры в обработанном числе.

В заключение рассмотрим пример употребления команды ED. Пусть на поле N длиной 4 находится десятичное упакованное число неизвестного знака, старшая цифра которого заведомо ноль. Мы хотим приготовить это число для печати в литерном виде на поле P длиной 11 по следующей схеме: знак числа, пробел, три цифры, запятая, три младшие цифры, два пробела. Для этого мы воспользуемся шаблоном, который в принятых ранее обозначениях имеет вид:

```
          *ЗЦЦЦ,ЦЦЦ**
          MVC   P(11),=X'402120202069202020404040'  шаблон
          ED    P(11),N                             число без
*                                                   знака
          MVI   P,C'+'                              занесение
*                                                   знака
                                                    плюс
*         BNL   OUT
          MVI   P,C'-'                              исправле-
*                                                   ние знака
OUT       ...
```

Код X'40' есть код пробела, а код X'69' — код запятой.

Глава 5

ПОДПРОГРАММЫ И ПРОГРАММНЫЕ МОДУЛИ, СТРУКТУРА МАШИННОЙ ПРОГРАММЫ

§ 5.1. Процедуры

Составление сколько-нибудь длинной программы на алгоритмическом языке высокого уровня практически невозможно без использования аппарата процедур. Описание частей алгоритма в виде процедур позволяет, во-первых, сократить длину программы, во-вторых, делает ее структуру более наглядной и, в-третьих, позволяет проводить ее отладку по частям.

Сказанное в полной мере относится и к программированию на машинно-ориентированном языке, где аналогом процедуры является подпрограмма.

Способы реализации процедур можно разделить на две группы. К первой относятся методы компиляции, существо которых состоит в замене каждого оператора процедуры текстом соответствующей подпрограммы, измененным в зависимости от фактических параметров оператора. Другая группа методов, называемых методами интерпретации (или методами обращения), использует один экземпляр текста подпрограммы, заменяя каждый оператор процедуры группой команд, которые передают подпрограмме параметры конкретного оператора. Во всем дальнейшем изложении в настоящей главе мы будем говорить только о методе обращения.

В системе программирования ЕС ЭВМ приняты определенные соглашения, в рамках которых структура оператора обращения к подпрограмме и структура самой подпрограммы в общих чертах описываются следующей схемой, изображенной на рис. 23.

При этом стандартным образом используются общие регистры с номерами: 0, 1, 13, 14, 15. В дальнейшем мы будем называть эти регистры специальными (или регистрами связи), в отличие от собственных регистров, номера которых принадлежат диапазону 2—12. Регистры 0 и 1 используются для передачи параметров, регистр 13 содержит адрес области сохранения подпрограммы — той части области переменных подпрограммы, куда копируется состояние общих регистров в момент обраще-

ния. В регистре 14 запоминается адрес возврата из подпрограммы. Регистр 15 перед входом в подпрограмму загружается адресом точки входа, а при выходе из подпрограммы обычно со-

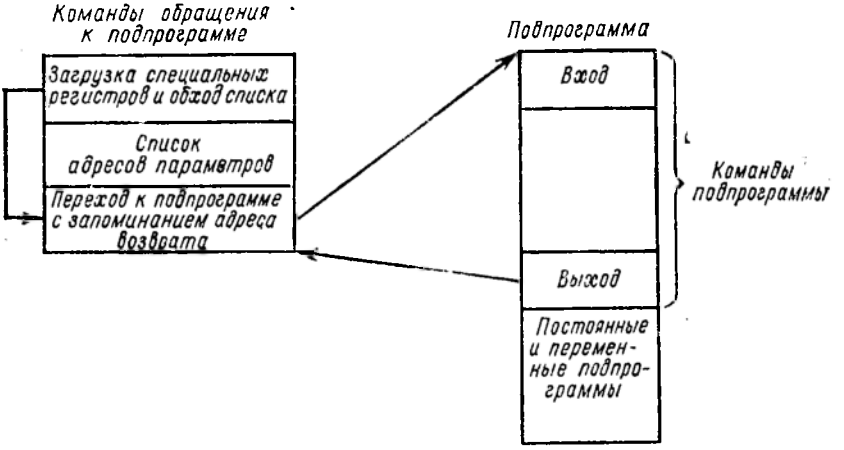


Рис. 23. Схема обращения к подпрограмме

держит код возврата — число, характеризующее результат исполнения подпрограммы.

§ 5.2. Подпрограммы без параметров

Рассмотрим сначала такие подпрограммы, при написании которых известно расположение обрабатываемой информации, и поэтому обращение к ним содержит только команду перехода. По аналогии с процедурами в смысле АЛГОЛ-60 назовем их подпрограммами без параметров. Будем также считать алгоритмы этих подпрограмм настолько простыми, что они могут использовать общие регистры совместно с собственно программой.

Для перехода к подпрограмме употребляется одна из двух машинных команд:

BAL R_1, $D_2(X_2, B_2)$
BALR R_1, R_2

Команда BAL (Branch And Link) формата RX осуществляет безусловный переход по адресу $D_2(X_2, B_2)$, загружая при этом в регистр R_1 адрес байта *, непосредственно следующего за командой. Команда BALR формата RR осуществляет безусловный

* Точнее, в регистр поступает вторая половина PSW, которая кроме адреса следующей команды в трех последних байтах в старшем байте содержит код длины, значение CC и маску программных прерываний.

переход по адресу, хранящемуся в регистре R_2, а регистр R_1 изменяет так же, как команда BAL.

При входе в подпрограмму без параметров никакие специальные действия не нужны, а выход состоит из команды безусловного перехода в точку возврата.

Поясним сказанное простым примером. Составим подпрограмму READ, которая при каждом обращении находит во входной строке, воспринимаемой с перфокарт, очередной символ, отличный от пробела, указывая его адрес в регистре 2. Если же при очередном обращении окажется, что перфокарты исчерпаны, подпрограмма осуществляет переход в точку ENDINF.

Очевидно, что подпрограмма должна обладать рабочим полем в 80 байтов (назовем его RF) для размещения части строки, занимающей одну перфокарту. При каждом обращении нужно либо продвигать адрес в регистре 2 по полю RF, либо читать следующую карту, возвращая регистр 2 в начальное состояние. Подпрограмму можно написать так:

```
READ    C     2, MAX        не исчерпана ли очередная
*                           перфокарта?
        BL    TEST          переход, если карта не ис-
*                           черпана
*       GET   IN, RF        чтение следующей карты
        LA    2, RF — 1     начальный адрес для сле-
*                           дующей перфокарты
TEST    LA    2, 1(2)       продвижение адреса на
*                           один байт
        CLI   0(2), C' '    если пробел, то
        BE    READ          на выборку следующего
*                           символа
        BR    3             выход из подпрограммы
MAX     DC    A(RF + 79)    максимальный адрес на
*                           поле RF
RF      DC    80X'00'       поле для очередной перфо-
*                           карты
```

Наша подпрограмма использует блок IN для управления вводом информации с перфокарт, а адрес очередного символа сообщает в общем регистре 2. Из ее структуры ясно, что перед первым обращением к ней необходимо загрузить в регистр 2 адрес, бо́льший или равный RF + 79. Для запоминания адреса возврата используется собственный регистр 3. Регистр 14 для этой цели употребить нельзя, так как его занимает макрокоманда GET

В этом примере мы воспользовались командой ассемблера DC с операндом вида

которая формирует* в машинном слове адресную постоянную — двоичное представление машинного адреса, эквивалентного переместимому выражению, указанному в скобках на поле операндов. Подробнее об адресных постоянных см. [5], с. 156, а также § 5.5. настоящего пособия.

Программа в целом (назовем ее MAIN), включающая в себя READ, должна иметь следующую организацию:

```
MAIN    PROC    12
        OPENIN  IN, ENDINF      открытие блока упра-
*                               вления вводом
        LA      2, RF+80        начальная загрузка
*                               регистра 2
        ...
        BAL     3, READ         обращение к READ
        ...
        BAL     3, READ         обращение к
        ...
        ...
        EXIT
READ    ...                     текст подпрограммы
*                               READ
        END
```

§ 5.3. Подпрограммы с параметрами

В языках высокого уровня параметры оператора процедуры могут иметь разную синтаксическую форму: идентификатор, переменная с индексами, выражение. На машинном языке параметр обычно задается адресом, указывающим на первый байт соответствующей информации, которая может быть числом, вектором, подпрограммой и т. д. В системе программирования ЕС ЭВМ для передачи параметров в общем случае используется схема, изображенная на рис. 24.

В общем регистре 1 указывается адрес списка адресов параметров, состоящего из группы соседних машинных слов. Каждое слово списка в трех последних байтах содержит начальный адрес соответствующего параметра. Если подпрограмма имеет переменное количество параметров, в первом байте последнего элемента списка содержится код X'80', а в первых байтах остальных элементов — коды X'00'. Иногда для передачи

---

* На самом деле ассемблер лишь подготавливает информацию для формирования адресной постоянной при загрузке в память приготовленной для исполнения машинной программы.

параметров используется также и регистр 0. Список адресов параметров может располагаться произвольно, но стандартное его место — между командами обращения к подпрограмме.

Для образования списка адресов и написания команд обращения можно пользоваться системной макрокомандой CALL, имеющей следующий формат:

$$[\text{имя}] \quad \text{CALL} \quad \left\{\begin{matrix}\text{имя точки входа}\\(15)\end{matrix}\right\} \left[,\begin{matrix}(\text{список}\\\text{параметров})\end{matrix} \left[,VL\right]\right]$$

Имя точки входа задается обычно переместимым выражением. Только в том случае, когда автор сам позаботился о загрузке адреса входа в регистр 15, можно написать число 15 в круглых

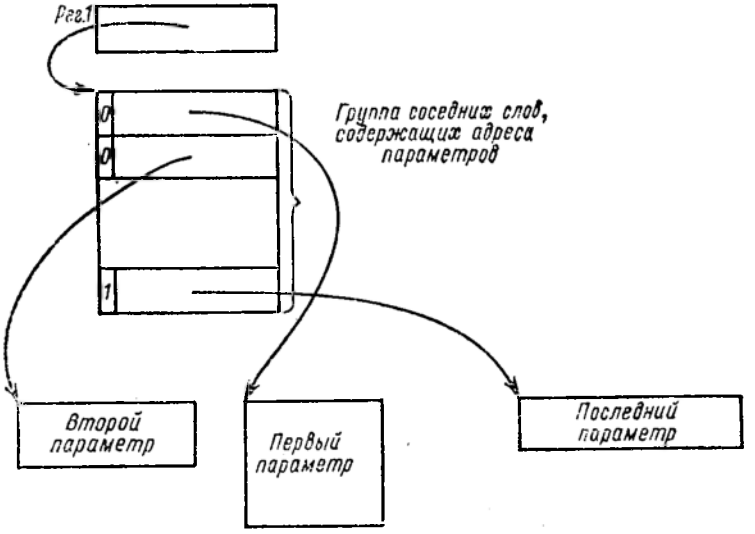


Рис. 24. Схема передачи параметров подпрограмме

скобках. Список параметров содержит переместимые выражения, эквивалентные соответствующим адресам, или, если эти адреса хранятся в собственных регистрах, номера соответствующих регистров в скобках. Параметр VL пишется в том случае, когда мы хотим отметить последний элемент списка кодом X'80'.

Пусть, например, нужно обратиться к подпрограмме SUBPRG с тремя параметрами, адреса которых соответственно A, (5), B — 20 (т. е. адрес второго параметра хранится в регистре 5). Если мы напишем макрокоманду:

```
CALL    SUBPRG, (A, (5), B—20)
```

ассемблер заменит ее следующей (с точностью до обозначений) последовательностью команд:

```
        L       15, L2      загрузка адреса входа
        ST      5, L1       пересылка адреса второго
*                           параметра
        CNOP    0, 4        выравнивание на границу
*                           слова
        BAL     1, L3       загрузка адреса списка в
*                           регистр 1 и переход в точ-
*                           ку L3
        DC      A(A)        список
L1      DS      F           адресов
        DC      A(B—20)     параметров
L2      DC      V(SUBPRG)   адрес входа в подпро-
*                           грамму
L3      BALR    14, 15      переход к подпрограмме
```

Команда ассемблера CNOP (Conditional No OPeration) вставляет в текст программы, если нужно, пустую машинную команду

```
        BCR     0, 0
```

с таким расчетом, чтобы следующая команда

```
        BAL     1, L3
```

разместилась в границах машинного слова. Если этого не сделать, между этой командой и первым словом списка адресов может образоваться промежуток в два байта и адрес в регистре 1 не будет совпадать с началом списка.

Команда ассемблера

```
        DC      V(SUBPRG)
```

образует адресную постоянную внешнего типа, используемую для связи между программными модулями (см. § 5.5).

В связи с некоторыми особенностями адреса типа V необходимо также добавить к программе информационную команду ассемблера

```
ENTRY   SUBPRG
```

Перейдем теперь к изучению структуры подпрограмм. В общем случае программа может содержать несколько подпрограмм: A, B, C и т. д. При этом подпрограмма A, начав выполняться, может, в свою очередь, обратиться к подпрограмме B. Такая цепочка вызовов может быть сколь угодно длинной. Рассмотрим схему, изображенную на рис. 25.

Собственно программа M обращается к подпрограмме A, которая, не закончив работы, обращается к подпрограмме B.

Программа М по отношению к управляющей программе операционной системы также является подпрограммой, вызываемой по общим правилам, и должна по окончании работы осуществить переход к ней по адресу, оставленному управляющей программой в регистре 14. Подпрограммы М, А, В могут писаться и отлаживаться разными людьми в разное время. Так как все они используют для связи и для других целей одни и те же общие регистры 0—15, необходимо договориться о средствах защиты информации в этих регистрах, принадлежащей одной подпрограмме, от действий другой подпрограммы. Мы опишем соглашения, принятые в системных программах ЕС ЭВМ. Настоятельно рекомендуем придерживаться этих же соглашений

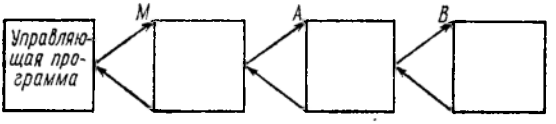


Рис. 25. Последовательный вызов подпрограмм

во всех пользовательских программах. В противном случае употребление системных макрокоманд и средств отладки станет невозможным.

Каждая подпрограмма должна иметь в своей переменной части поле в 18 машинных слов, называемое областью сохранения подпрограммы. В этой области запоминаются значения, оставленные данной подпрограммой в общих регистрах, на время работы вызванной подпрограммы.

Каждая подпрограмма с точки зрения обращения может быть вызывающей (когда она обращается к другой подпрограмме) или вызываемой (когда к ней обращается другая подпрограмма). По соглашению именно вызываемая подпрограмма должна записать в область сохранения вызывающей подпрограммы значения из всех общих регистров, которые она может изменить, а при возвращении в вызывающую подпрограмму восстановить все измененные значения. Расположение информации в областях сохранения SM, SA и SB подпрограмм М, А и В приведено на рис. 26. Первое слово каждой области сохранения является резервным и может использоваться произвольно.

Вызывающая подпрограмма обязана до перехода к вызываемой загрузить в общий регистр 13 адрес первого слова своей области сохранения. Стандартно эта загрузка производится при входе в подпрограмму. Вызываемая подпрограмма должна записать во второе слово своей области сохранения адрес области сохранения вызывающей подпрограммы, а в третье слово области сохранения вызывающей подпрограммы — адрес своей области сохранения. Так, подпрограмма А должна в своем входе иметь команды записи адреса SM в SA + 4, адреса SA в SM +

+ 8 и загрузки адреса SA в регистр 13. Вход в подпрограмму B должен содержать команды, загружающие в слова с номерами от 4 до 18 области SA значения тех регистров, которые подпрограмма B может изменить (а также, конечно, команды записи SB в SA + 8 и SA в SB + 4). Безопаснее всего запоминать все регистры — от 14 до 12 — одной командой STM кратной выгрузки. Выход из подпрограммы B должен содержать команды восстановления регистра 13 (по второму слову в SB) и всех остальных измененных регистров (по информации в SA), за исключением, может быть, лишь регистра 15, в который был записан

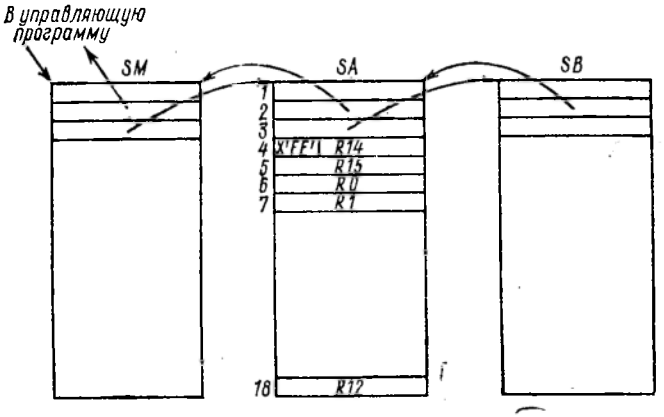


Рис. 26. Области сопряжения последовательно вызываемых подпрограмм

код возврата, характеризующий результат исполнения подпрограммы B.

Вторые и третьи слова областей сохранения цепочки последовательно вызываемых подпрограмм образуют список с двумя связями. Вход сверху в этот список находится в области сохранения управляющей программы, а снизу — в регистре 13, занятом последней подпрограммой цепочки. При аварийной ситуации сервисные программы операционной системы по этой информации находят и печатают содержимое областей сохранения, что позволяет установить, в какой подпрограмме произошла неприятность. Для облегчения отладки полезно также при выходе из подпрограммы в старшем байте четвертого слова области той подпрограммы, в которую происходит возврат, делать отметку кодом X'FF' о том, что возврат имел место.

Описанные выше действия при входе в подпрограмму B можно обеспечить следующими командами:

```
B       STM     14, 12, 12(13)      запись в SA содержимого
*                                   регистров
        LR      5, 13               копирование адреса SA в
*                                   регистр 5
```

```
         LA      13, SB          загрузка в регистр 13 адреса SB
*
         ST      13, 8(5)        запись адреса SB в третье слово области SA
*
         ST      5, 4(13)        запись адреса SA во второе слово области SB
*
         ...
SB       DC      18F'0'          область сохранения подпрограммы B
*
```

Все эти действия, включая резервирование 18 слов для области сохранения, обеспечиваются системной макрокомандой PROC, которая может быть написана в одном из следующих видов:

```
B        PROC
```

или

```
B        PROC    RENT=NO
```

В первом случае область сохранения динамически резервируется заново при каждом входе в подпрограмму, во втором — строится с помощью команды ассемблера DC как часть расширения макрокоманды. Второй вариант для исполнения требует меньше машинного времени, но не пригоден для рекурсивных и повторно входимых подпрограмм (см. § 5.4 и 5.6).

Действия, необходимые при выходе из подпрограммы B, могут осуществляться следующими командами:

```
         L       13, 4(13)       запись в регистр 13 адреса SA
         LM      14, 12, 12(13)  восстановление регистров 14 —
*                                —12
         MVI     12(13), 255     отметка о возврате в четвертом
*                                слове
         BR      14              возврат в A
```

Эти же действия осуществляются системной макрокомандой

```
         EXIT    REGS=(14, 12)
```

Отличие состоит лишь в том, что в регистр 15 вместо его восстановления по пятому слову SA заносится 0. Если в макрокоманде опустить параметр REGS, то восстановятся лишь регистры связи 13 и 14 и все собственные регистры.

В качестве примера рассмотрим подпрограмму DIV, обращение к которой имеет форму

```
         CALL    DIV, (A, B, C, D)
```

Подпрограмма должна разделить десятичное упакованное число A на такое же B, поместив частное на поле C, а остаток — на

поле D. Предполагается, что A, B, C и D имеют длину $l \leqslant 8$, двоичное значение которой служит пятым параметром подпрограммы DIV и должно быть помещено в общий регистр 0 перед обращением к подпрограмме.

Обозначив через W поле длины $L = 2l$, мы можем описать действия, которые должна обеспечить подпрограмма, следующими командами:

ZAP	W(L), A(l)	запись делимого в W
DP	W(L), B(l)	частное и остаток в W
ZAP	C(l), W(l)	запись частного в C
ZAP	D(l), W + 1(l)	запись остатка в D

Но так как адреса A, B, C, D и число l во время написания программы неизвестны, для реализации нужных действий можно поступить следующим образом:

вычислить адреса A, B, C, D, W + 1 в общих регистрах 2, 3, 4, 5, 6;

вычислить пары длин (l-1, l-1) и (L-1, l-1) в последних байтах общих регистров 7 и 8 (по четыре бита на каждую компоненту пары);

воспользоваться командами EX.

```
DIV     PROC    RENT=NO     вход в подпрограмму DIV
*
        LM      2,5,0(1)    адреса A, B, C, D в регистрах 2—5
*
        LA      6,W
        AR      6,0         адрес W+1 в регистре 6
*
        BCTR    0,0         l — 1 — машинная длина в регистре 0
*
        LR      7,0         l — 1 в регистре 7
        SLA     0,4         (l — 1, 0) в регистре 0
        AR      7,0         (l — 1, l — 1) в регистре 7
*
        LR      8,7         (l — 1, l — 1) в регистре 8
*
        AR      8,0         (L — 2, l — 1) в регистре 8
*
        LA      8,16(8)     (L — 1, l — 1) в регистре 8
*
        EX      8,ZAPWA     W := A
        EX      8,DWB       W := A ÷ B;
*                           W + 1 := остаток
        EX      7,ZAPCW     C := W(l)
```

```
         EX      7, ZAPDW        D:=W+1(1)
         EXIT    REGS=(0, 8)     восстановление регистров 0—8, 13, 14
*
ZAPWA    ZAP     W(0), 0(0, 2)
DWB      DP      W(0), 0(0, 3)
ZAPCW    ZAP     0(0, 4), W(0)
ZAPDW    ZAP     0(0, 5), 0(0, 6)
W        DS      16C             поле для максимальных операндов
*
```

§ 5.4. Рекурсивные подпрограммы

Мы знаем, что при исполнении подпрограммы могут возникать цепочки последовательных вызовов. При этом возможна следующая ситуация: подпрограмма A обратилась к подпрограмме B, которая, в свою очередь, обратилась опять к подпрограмме A. Такое обращение подпрограммы A к себе самой через посредство другой подпрограммы (или целой цепочки подпрограмм) называется неявной рекурсией. На практике чаще встречается явная рекурсия, когда в теле подпрограммы A находятся команды обращения к ней самой. Ясно, что обычным образом построенная подпрограмма неспособна к рекурсивному использованию: при повторном обращении к ней до завершения предыдущего информация о незавершенном предыдущем обращении, хранившаяся в ее переменных и области сохранения регистров, может быть невозвратимо утрачена. Чтобы этого избежать, нужно при каждом рекусивном обращении выделять для области сохранения и переменных новый участок памяти. Механизм, реализующий такое размножение переменной части подпрограммы, может быть различным. Например, можно динамически резервировать с помощью макрокоманды GETMAIN при каждом входе в подпрограмму новый участок памяти для области сохранения и переменных, используя регистр 13 в качестве базисного для переменных.

Пусть в подпрограмме A фигурируют следующие переменные: двойное машинное слово P, слово Q и текст T длиной 15 байтов. Тогда для области сохранения (18 слов) и переменных потребуется 99 байтов. Вход в A можно написать так:

```
A        STM     14, 12, 12(13)  запоминание регистров
*                                вызывающей подпро-
*                                граммы
         GETMAIN R, LV=104       заказ участка в
*                                104 байта
         ST      13,4(1)         связь областей
         ST       1,8(13)        сохранения
```

```
         LM      15,1,16(13)     восстановление регистров
*                                15,0,1, измененных макро-
*                                командой GETMAIN
         L       13, 8(13)       адрес новой области
```

Переменные мы разместим непосредственно за последним словом области сохранения. Тогда P, Q, и T будут соответственно определяться следующими явными выражениями: 72 (13), 80 (13), 84 (13), с помощью которых можно из тела подпрограммы ссылаться на эти переменные *.

Если вход реализован только что описанным способом, новый участок памяти резервируется при каждом обращении к подпрограмме. Это может привести к неоправданному расходу памяти в тех случаях, когда цепочка рекурсивных обращений к подпрограмме A повторяется многократно. Чтобы этого избежать, можно при каждом выходе освобождать соответствующий участок с помощью макрокоманды FREEMAIN. Напишем выход из A следующим образом:

```
         LR      1, 13           адрес начала уча-
*                                стка в регистре 1
         FREEMAIN R, LV=104, A=(1) освобождение уча-
*                                стка в 104 байта
         L       13, 4(13)       восстановление ре-
*                                гистра 13
         LM      14, 12, 12(13)  восстановление ре-
*                                гистров 14—12
         BR      14              возврат к вызыва-
*                                ющей подпро-
*                                грамме
```

Теперь при каждом выходе из подпрограммы A будет освобождаться участок памяти, занятый при соответствующем входе, и общее количество участков, одновременно занятых подпрограммой, будет равно количеству незавершенных обращений.

Только что предложенный вариант выхода иногда может при исполнении задачи в мультипрограммном режиме привести к ошибке. Дело в том, что сообщение об освобождении участка поступает в систему до использования хранящейся в нем информации. Может оказаться, что наша задача будет приостановлена как раз после выполнения FREEMAIN и освобожденный участок будет передан другой задаче, которая испортит информацию

---

* Если в теле подпрограммы воспользоваться командами ассемблера USING и DSECT, можно ссылаться на эти переменные с помощью ассемблерных имен. См. по этому поводу § 5.6.

о состоянии регистров. Избежать эту неприятность можно, восстанавливая регистры до освобождения области:

```
        LR      1, 13           копирование адре-
*                               са из регистра 13
        L       13, 4(13)       восстановление ре-
*                               гистра 13
        L       14, 12(13)      восстановление ре-
*                               гистра 14
        LM      2, 12, 28(13)   восстановление ре-
*                               гистров 2—12
        FREEMAIN R, LV=104, A=(1) освобождение
*                               участка
        BR      14              возврат в точку
*                               вызова
```

В этом варианте, поскольку команда FREEMAIN меняет значения в регистрах 0, 1 и 15, названные регистры не будут восстановлены.

В частном случае все переменные рекурсивной подпрограммы могут размещаться в ее собственных регистрах. Тогда при входе достаточно размножать лишь область сохранения. Эту задачу автоматически решает макрокоманда PROC без параметров или с ключевым параметром TM = NO. В последнем случае участок в 18 слов динамически резервируется заново при каждом входе в подпрограмму. В первом же случае (т. е. при отсутствии ключевых параметров) производится экономия памяти следующим образом. Машинные команды, образующие расширение PROC, исследуют код третьего по порядку слова области сохранения, адрес которой был передан в регистре 13. Если там обнаружен код, содержащий нули не во всех своих позициях, то новая область не резервируется, а в регистр 13 заносится адрес из третьего слова предыдущей области. При таком способе выделение нового участка в 18 слов происходит столько раз, какова максимальная глубина рекурсии. Например, рассматриваемая ниже подпрограмма HT при n = 10 вызывается 1023 раза, а выделение нового участка происходит всего 10 раз.

Естественно, такой способ может быть использован лишь при строгом соблюдении всех соглашений об областях сохранения. В частности, при выделении новой области для переменных и регистров нужно очищать всю заказанную область или хотя бы ее третье слово. Макрокоманда PROC автоматически производит эти действия.

Рассмотрим в качестве примера рекурсивную процедуру, печатающую последовательность ходов в игре «Ханойские башни»*. Игра состоит в следующем. Имеются три колышка с но-

* Эта процедура приведена качестве примера в официальном описании языка АЛГОЛ-68.

мерами 1, 2, 3. На колышек с номером p надета стопка колец разных диаметров, причем диаметры монотонно увеличиваются в направлении сверху вниз. Нужно последовательностью ходов перенести всю стопку на колышек q, не изменив взаимного расположения колец. Один ход состоит в переносе верхнего кольца с одного из колышков на свободный или уже занятый, причем в последнем случае необходимо, чтобы переносимое кольцо легло на кольцо бо́льшего диаметра.

Решение задачи с n кольцами очевидным образом сводится к двум задачам переноса верхней части стопки из $n-1$ кольца и к еще одному ходу. Рекурсивное описание соответствующей процедуры на АЛГОЛ — 60 таково:

procedure HT (n, p, q); **value** n, p, q; **integer** n, p, q;
if n = 1 **then** print (p, q) **else begin** HT (n — 1, p, 6 — p — q);
print (p, q); HT (n — 1, 6 — p — q, q) **end**

Оператор print (p, q) печатает один ход в виде пары целых чисел, первое из которых означает номер колышка, с которого снимается верхнее кольцо, а второе — номер того, на который кольцо надевается.

Реализуем эту поцедуру в виде рекурсивной подпрограммы без параметров, предполагая, что при каждом обращении к ней значения n, p, q располагаются в виде двоичных чисел в регистрах соответственно 2, 3, 4. Для печати хода воспользуемся системной макрокомандой SNAP, печатающей значения из всех регистров. Числа p и q, определяющие очередной ход, в момент печати будут находиться в регистрах соответственно 3 и 5. Текст подпрограммы может быть таким:

```
HT      PROC    ,  вход в рекурсивную подпрограмму
        LR      5, 4 запоминание q в регистре 5
        BCT     2, ELSE  n:=n — 1; if n > 0 then go to
*               ELSE
        SNAP    DCB = OUT, PDATA = REGS  print (p, q)
        B       EXIT     на выход из HT
ELSE    LA      4, 6          R4:=6
        SR      4, 3          R4:=6 — p
        SR      4, 5          R4:=6 — p — q
        BAL     14, HT HT (n — 1, p, 6 — p — q)
        SNAP    DCB = OUT, PDATA = REGS print (p, q)
        LR      3, 4          R3:=6 — p — q
        LR      4, 5          R4:=q
        BAL     14, HT  HT (n — 1, 6 — p — q, q)
EXIT    EXIT
```

Имя OUT в подпрограмме означает адрес блока управления печатью.

Включим нашу подпрограмму в программу, которая печатает решение задачи с n = 10, p = 1, q = 2:

```
PRG     PROC        12, RENT=NO     вход в программу
*                                   PRG
        OPENSNAP    OUT             открытие блока уп-
*                                   равления печатью
        LA          2, 10           n := 10
        LA          3, 1            p := 1
        LA          4, 2            q := 2
        BAL         14, HT          внешнее обраще-
*                                   ние к HT
        EXIT        ,               выход из програм-
*                                   мы PRG
HT      PROC        ,               вход в HT
        ....
EXIT    EXIT        ,               выход из HT
        END         ,               конец программы
*                                   PRG
```

Все переменные подпрограммы находятся в четырех собственных регистрах: n — в регистре 2, p — в регистре 3, q и 6—p—q — в регистре 4, q — в регистре 5. Из регистров связи

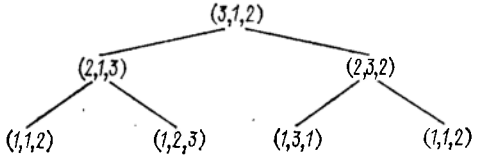


Рис. 27. Дерево вызовов подпрограммы HT при исполнении оператора HT (3, 1, 2)

используются только 13—15. Для решения задачи с n кольцами подпрограмма HT вызывается, как нетрудно подсчитать $2^n - 1$ раз. При этом длина цепочки рекурсивных вызовов составляет n. При n = 3 дерево вызовов изображено на рис. 27, где числа в круглых скобках определяют значения n, p и q при каждом вызове. Последовательность ходов при n = 3:

(1, 2) (1, 3) (2, 3) (1, 2) (3, 1) (3, 2) (1, 2).

§ 5.5. Программные модули и структура машинной программы

До сих пор, говоря о программе, предназначенной для исполнения на машине в качестве пункта задания, мы считали, что она состоит из собственно программы и нескольких под-

программ, которые вызываются собственно программой или обращаются друг к другу. Пример организации такой программы содержит схема:

```
PRG     PROC    12, RENT=NO
        команды собственно программы
            EXIT    ,                   выход из PRG
        постоянные и переменные PRG
A       PROC    RENT=NO                 вход в подпрограмму A
        команды подпрограммы A
            EXIT    ,                   выход из A
        постоянные и переменные подпрограммы A
B       PROC    RENT=NO                 вход в подпро-
*                                       грамму B
        команды подпрограммы B
            EXIT    ,                   выход из подпро-
*                                       граммы B
        постоянные и переменные подпрограммы B
            END
```

Здесь предполагается, что для адресации элементов собственно программы и подпрограмм A и B, указанных перемещаемыми выражениями, ассемблер использует регистр 12 с одним и тем же базисным адресом. Поэтому разделение переменных и постоянных, относящихся к собственно программе и подпрограммам A и B, носит формальный характер. Например, подпрограмма B может использовать постоянные A на таких же правах, как свои собственные. Такая тесная связь имеет ряд неудобств. Для отладки, скажем, подпрограммы B мы должны выполнять ее либо в составе PRG, либо превратив в самостоятельную программу. В первом случае нужно вводить в машину много лишней информации, а во втором — вносить в текст B ряд изменений, которые, в свою очередь, могут содержать ошибки. Более удобным путем конструирования большой программы является компоновка ее из равноправных, как можно меньше зависящих друг от друга частей — программных модулей, которые можно писать, отлаживать и хранить отдельно друг от друга и только в последний момент объединять в программу, предназначенную для исполнения процессором вычислительной системы. Структура программного модуля зависит от операционной системы и входного языка программирования. Мы в дальнейшем будем говорить о модулях ассемблера, предназначенных для исполнения в рамках операционной системы ОС ЕС ЭВМ.

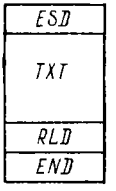


Рис. 28. Структура объектного модуля

Программный модуль — это знакомая нам подпрограмма, оформленная по определенному стандарту. В ОС ЕС ЭВМ различаются четыре вида модулей: входной, объектный, загрузочный и рабочий. Входной модуль представляет собой текст на языке ассемблера, особенности которого мы подробно рассмотрим ниже. Он составляет входную информацию для компилятора ассемблера, перерабатывающего его в объектный модуль. Схема последнего приведена на рис. 28. Основную часть модуля (TXT) составляет последовательность машинных команд, постоянных и переменных, расположенных в том порядке, в каком были расположены их прообразы во входном модуле. Каждая команда представлена в машинной форме: код операции, номера регистров, смещения, непосредственные операнды замены соответствующими двоичными кодами. Все постоянные переведены в ту форму, в какой они могут восприниматься процессором при выполнении программы, а места переменных, соответствующих командам DS входного модуля, заполнены случайной информацией. Весь текст разбит на записи длиной в 80 байтов, причем собственно текст занимает в каждой записи байты с номерами от 17 до 72, а остальные байты содержат служебную информацию.

Часть ESD(External Symbol Dictionary) — словарь внешних связей модуля. Он содержит ассемблерные имена и указания о расположении всех тех точек модуля, на которые возможны ссылки из других модулей, а также перечень всех имен в других модулях, ссылки на которые содержит данный модуль.

Часть RLD (ReLocation Dictionary) — словарь перемещений — содержит информацию об адресных постоянных модуля. Эти постоянные не могут быть определены ассемблером, так как их значения зависят от места расположения модуля в оперативной памяти при его исполнении. Если словарь перемещений пуст, текст модуля представляет переместимую программу, которая без каких-либо изменений может исполняться в любом месте оперативной памяти. Дальнейшие подробности о структуре объектного модуля читатель может найти в руководствах [9], [15] или [16].

Структура загрузочного модуля подобна структуре объектного. Он тоже состоит из текста, словаря внешних ссылок, словаря перемещений. Отличие в том, что текст плотно упакован в записи переменной длины и на внешних носителях занимает гораздо меньше места, чем текст объектного модуля. Загрузочные модули хранятся на диске или магнитных лентах в библиотечных наборах данных, элементы оглавления которых, кроме имени модуля и указания на его место в библиотеке, содержат информацию о некоторых свойствах модуля (см. § 5.6). Загрузочные модули вырабатываются системной порграммой, называемой редактором связей (LinKage EDitor). В простейшем случае входной информацией для редактора служит один объектный

модуль. Редактор связей может также объединить в один загрузочный модуль несколько объектных и загрузочных модулей, разрешая межмодульные ссылки и объединяя друг с другом словари ESD и RLD поступивших на его вход модулей. Действия редактора связей программируются указанием его параметров (Options), управляющими операторами (INCLUDE, NAME, ENTRY и др.) и DD-предложениями, определяющими входные и выходные наборы данных. Подробнее о работе этой сложной программы см. в [16].

Загрузочный модуль, не содержащий в своем ESD ссылок на другие модули, для превращения в рабочий модуль, который может исполняться процессором системы, нуждается лишь в определении перечисленных в RLD адресных постоянных. Эта последняя операция производится системной программой выборки непосредственно перед загрузкой модуля в отведенное для него место оперативной памяти. С этого момента модуль становится неперемещаемым и не может загружаться в другое место.

Перейдем теперь к рассмотрению структуры входного модуля, представляющего текст на языке ассемблера. Входной модуль начинается командой ассемблера:

```
имя     CSECT     комментарий
```

До сих пор мы не пользовались этой командой только потому, что она автоматически включается в расширение системной макрокоманды PROC, содержащей в качестве позиционного параметра номер базисного регистра. Конец модуля обычно определяется командой

```
        END       [имя точки входа]
```

Имя, расположенное на поле имени команды CSECT, всегда заносится в словарь ESD объектного модуля и не должно фигурировать в командах ENTRY (см. ниже). Так как команда CSECT не имеет образа в объектном модуле, это имя указывает на следующую команду. По умолчанию оно считается точкой входа в модуль (т. е. в том случае, когда в поле операндов заключительной команды END не указано другое имя).

Каждый модуль в нормальном случае имеет по крайней мере один базисный регистр — отличный от нулевого общий регистр, через который компилятор ассемблера адресует неявно указанные операнды. Автор программы назначает базисный регистр (или регистры) с помощью команды ассемблера

```
        USING     перемещаемое выражение, список базисных
                  регистров
```

Например, команда

```
        USING     A, 12, 11
```

информирует ассемблер, что адрес, соответствующий имени A, хранится в регистре 12, а адрес, соответствующий A + 4096, находится в регистре 11. Так как ассемблер просматривает модуль в порядке сверху вниз, эта информация доступна для адресации лишь в командах, расположенных ниже команды USING. Поэтому последнюю обычно помещают в начале модуля. Загрузка базисных регистров производится машинными командами, входящими в состав модуля, и должна осуществляться раньше (в порядке исполнения команд), чем будут исполняться команды, использующие базисный регистр. Поэтому команды загрузки располагаются вблизи от точки входа. Обычно для этой цели применяется команда BALR. Например, если точка входа в модуль M совпадает с его началом и мы хотим назначить в качестве базисного регистр 12, модуль можно начинать так:

```
M       CSECT
        STM     14, 12, 12(13)      сохранение регистров
        USING   *+2, 12             объявление базисного
*                                   регистра 12
        BALR    12, 0               загрузка базисного ре-
*                                   гистра 12
```

Если мы уверены, что переход к модулю будет всегда осуществляться через регистр 15, модуль можно начинать так:

```
M       CSECT
        STM     14, 12, 12(13)      сохранение регистров
        USING   M, 12               объявление базисного
*                                   регистра 12
        LR      12, 15              перепись адреса входа
*                                   в регистр 12
```

Не следует использовать в качестве базисного регистр 15, так как его значение изменяется системными макрокомандами.

Если модуль настолько велик, что одним базисным регистром не обойтись, можно назначить базисными два или больше регистров. Например, в модуле N для этой цели служат регистры 11 и 12:

```
N       CSECT
        STM     14, 12, 12(13)      сохранение регистров
        BALR    12, 0               загрузка в регистр 12
*                                   адреса A
        USING   *, 12, 11           объявление A в ре-
*                                   гистре 12 и A+4096
*                                   в регистре 11
```

```
A       L       11, BA11        загрузка в регистр 11
*                               адреса A + 4096
        B       BA11 + 4        обход адресной постоян-
*                               ной
BA11    DC      A(A + 4096)
```

Ссылки на адреса от A до A + 4095 могут осуществляться через регистр 12, а на большие (но не превосходящие A + 8191) — через регистр 11.

Заметим, что три последние команды начала модуля нельзя заменить одной такой:

```
        LA      11, A + 4096
```

Действительно, согласно объявлению в команде USING и тому, что смещение не должно превышать числа 4095, адрес A + 4096 можно выразить лишь через регистр 11 со смещением 0. Поэтому написанная выше команда эквивалентна такой:

```
        LA      11, 0(11)
```

что очевидно не имеет смысла. Но двумя командами LA загрузить регистр 11 можно, например, так:

```
        LA      11, A + 4000    загрузка адреса A+4000
        LA      11, 96(11)      увеличение адреса на 96
```

Области программы, адресуемые через разные базисные регистры, могут перекрываться. Программист может командой ассемблера

```
        DROP    список номеров регистров
```

отменять назначения базисных регистров, сделанные выше командами USING. Если какая-либо точка программы может адресоваться через несколько регистров, ассемблер выбирает тот, для которого требуется меньшее смещение. На следующей схеме показано, какие базисные регистры будет использовать ассемблер для неявных ссылок из различных точек программы A:

```
A       CSECT
        ....            }   нет доступных регистров
        USING   A, 12
        USING   B, 11
        ....            }
B       ....            }   регистр 12
        ....            }   регистр 11
        DROP    11
        ....            }   регистр 12
        END
```

При этом предполагается, что длина модуля A не превосходит 4096 байтов.

Поскольку каждый модуль является подпрограммой, он должен обладать областью сохранения регистров и включать в себя команды входа в подпрограмму. Модуль с одним базисным регистром и входом, совпадающим с началом, может начинаться макрокомандой

```
M       PROC        12, RENT=NO
```

Расширение этой макрокоманды выглядит (с точностью до некоторых деталей) следующим образом:

```
M       CSECT
        USING       M, 12           назначение базисного ре-
*                                   гистра 12
        STM         14, 12, 12(13)  запоминание регист-
*                                   ров 14—12
        LR          12, 15          загрузка базисного адреса
        CNOP        0, 4            пустая команда для выра-
*                                   внивания на границу слова
        BAL         1, *+76         запись в регистр 1 адреса
*                                   области сохранения M
        DC          18F'0'          область сохранения мо-
*                                   дуля M
        ST          13, 4(1)        обмен адресами областей
*                                   сохранения
        ST          1, 8(13)        с вызывающей подпро-
*                                   граммой
        L           1, 24(13)       восстановление регистра 1
        L           13, 8(13)       запись в регистр 13 адреса
*                                   области сохранения M
```

Макрокоманда PROC, начинающая модуль, должна иметь имя, а вход в такой модуль возможен только через регистр 15.

Выход из модуля не имеет никаких отличий от выхода из подпрограммы. Он может осуществляться с помощью макрокоманды EXIT.

Входной информацией для ассемблера могут служить несколько входных модулей, заканчивающихся одной командой END (но лишь в том случае, когда в этих модулях нет одинаковых ассемблерных имен). Тексты модулей могут перемежаться, как показано на следующей схеме:

```
A       PROC        12, RENT=NO
    начало текста модуля A
B       PROC        12, RENT=NO
    начало текста модуля B
```

```
A       CSECT
        продолжение текста модуля A
B       CSECT
        продолжение текста модуля B
        END
```

Пример такого рода представляет единственный случай, когда одно и то же имя фигурирует несколько раз на поле имени во входной программе. Ассемблер соберет вместе части модулей A и B и переработает их в объединенный объектный модуль.

Если входная программа состоит из нескольких модулей, ассемблер располагает все использованные в ней литералы в конце первого модуля, что делает невозможным ссылки на них не из первого модуля. Для того чтобы каждый литерал был отнесен к тому модулю, в котором он фигурирует, следует в области постоянных каждого модуля поместить команду ассемблера

```
        LTORG
```

(LiTerals ORiGin) с пустым полем операндов.

Как уже говорилось выше, модули служат теми элементами, из которых строится исполняемая машиной программа — некоторое множество модулей, связанных взаимными ссылками, предназначенное для решения определенной задачи. В операционной системе ОС ЕС ЭВМ в зависимости от типа допустимых связей различаются три вида программ: программы простой структуры, программы динамической структуры и программы с запланированным перекрытием. Мы сначала познакомимся с организацией простой структуры.

Модули, предназначенные для включения в простую структуру, могут иметь любую из трех форм: входную, объектную или загрузочную. Тексты модулей двух первых видов хранятся на перфокартах или в расположенных на дисках и магнитных лентах библиотеках входных и объектных модулей, а тексты загрузочных моделей — в библиотеках загрузочных модулей. Перед исполнением программы простой структуры все ее модули должны быть предварительно с помощью ассемблера и редактора связей объединены в один загрузочный модуль, не содержащий в своем ESD никаких внешних ссылок *. Главным модулем, с которого после загрузки в память объединенного модуля начинается исполнение программы, по умолчанию считается первый в порядке расположения. В противном случае нужно

* С помощью параметров NCAL и LET редактор связей может образовать выполнимый загрузочный модуль, содержащий в своем ESD ссылки на другие модули. Это имеет смысл в тех случаях, когда заведомо известно, что при исполнении модуля эти ссылки не будут использованы (см. [15, раздел 4.2.5]).

сообщить редактору связей имя точки входа в программу с помощью управляющего предложения ENTRY.

Поскольку взаимное расположение модулей простой структуры известно на стадии ее редактирования, между ними возможны разнообразные формы связи, запланированные при написании модулей с помощью команд ассемблера ENTRY (не путать с управляющим предложением!) и EXTRN. Именно, если в модуле A есть ссылки, например, на точки P, Q, R в модуле B, в модуле A должна быть команда

```
EXTRN     P, Q, R
```

а в модуле B команда

```
ENTRY     P, Q, R
```

которые могут находиться в произвольных местах соответствующих модулей *. Однако в некоторых случаях, оговоренных ниже, точки ссылки не должны фигурировать в этих командах.

Рассмотрим основные случаи связи в простой структуре.

а) Вызов из модуля A подпрограммы, оформленной в виде другого модуля B этой же программы.

Вызов производится с помощью макрокоманды CALL и по форме ничем не отличается от обращения к внутренней подпрограмме. Команды EXTRN и ENTRY в этом случае не нужны, так как в расширении макрокоманды CALL для перехода к B используется адресная постоянная внешнего типа, а в другом модуле его имя B автоматически заносится в его ESD.

б) Переход из модуля A во внутреннюю точку C модуля B. Этот переход может быть осуществлен по схеме:

```
A     CSECT              B     CSECT
      ...                      ENTRY C
      L 15, =V(C)              ...
      B 15               C     ...
      ...                      ...
      END                      END
```

При планировании переходов подобного рода надо быть очень осторожным: не забывать о своевременной загрузке базисных регистров, о сохранении и восстановлении информации в общих регистрах.

в) Ссылка из модуля A на постоянные или переменные модуля B.

---

* Команды ENTRY и EXTRN нужны лишь тогда, когда модули A и B по отдельности переводятся ассемблером в объектную форму и затем объединяются редактором связей в один загрузочный модуль.

В следующей схеме показано, как можно для перекодировки текста T, расположенного в модуле A, воспользоваться словарем, находящимся в модуле B:

```
A       CSECT                              B       CSECT
        EXTRN DICT                                 ENTRY DICT
        ...                                        ...
        L       5, =A(DICT)        DICT    .....
        TR      T, 0(5)                    END
        ...
T       ...
        END
```

Если мы хотим из модуля A ссылаться на некоторое множество постоянных или переменных модуля B, расположенных в последнем непосредственно друг за другом, с помощью ассемблерных имен, мы можем воспользоваться командой ассемблера

```
имя     DSECT     пусто
```

определяющей фиктивную программную секцию. Ассемблер не резервирует памяти для объектов, описанных в секции. Он только заменяет все неявные ссылки на эти объекты, сделанные в машинных командах модуля, через базисный регистр секции и соответствующее расстоянию объекта от начала секции смещение. Рассмотрим пример, поясняющий употребление команды DSECT. Пусть в модуле B имеется следующая последовательность описаний:

```
        DS      0D
C1      DC      F...
C2      DC      D...
C3      DC      15C...
C4      DC      E...
        ENTRY   C1
```

Построим вход в модуль A следующим образом:

```
A       PROC    12, RENT=NO
        EXTRN   C1
        USING   N, 11          назначение базисного ре-
*                              гистра фиктивной секции
        L       11, =A(C1)     загрузка базисного ре-
*                              гистра фиктивной секции
N       DSECT                  начало фиктивной секции
C1      DS      F              разметка
C2      DS      D              объектов, входящих в
C3      DS      15C            фиктивную
```

```
C4      DS        E                секцию
A       CSECT                      конец фиктивной секции
                  ....             продолжение текста модуля A
        END
```

В машинных командах, входящих в состав продолжения текста A, именами C1, C2, C3, C4 можно пользоваться почти на таких же правах, как в модуле B. Команда

```
        DS        0D
```

продвигающая счетчик адреса в модуле B на значение, кратное 8, необходима потому, что ассемблер предполагает именно такое выравнивание для начала фиктивной секции.

Модули A и B нашего примера не должны предъявляться ассемблеру в составе одной входной программы, так как содержат на поле имени одни и те же идентификаторы C1, C2, C3, C4.

г) Совместное употребление несколькими модулями общей области для переменных можно обеспечить с помощью команды ассемблера

```
пробел    COM       пусто
```

Эта область может быть в каждом модуле по-своему разбита на части командами ассемблера DS. Редактор связей, собирая модули в простую структуру, отводит в самом начале общего загрузочного модуля место размера, достаточного для размещения самой большой области. Общая область может описываться в каждом модуле подряд или по частям, перемежаемым частями программы модуля, подробно тому, как могут смешиваться два модуля во входной программе. При загрузке модуля в память общая область располагается с начала двойного слова. Никакой определенной информации в нее при этом не заносится. Пример организации общей области в модулях A и B содержит схема:

```
A       PROC      12, RENT=NO      B       PROC      12, RENT=NO
        USING     P, 11                    L         10, =A(R)
        L         11, =A(P)                USING     R, 10
        ...                                ...
        COM                                COM
P       DS        20C              R       DS        10C
Q       DS        30C              S       DS        20C
        END                        T       DS        5F
                                           END
```

---

* Имена, определенные в фиктивной секции, не должны использоваться для формирования адресных постоянных (см. [5, с. 191]).

Ссылки на P или Q в модуле A, на R, S или T в модуле B производятся обычным образом: перемещаемым выражением или указанием смещения и базисного регистра (11 в модуле A и 10 в модуле B). При этом ссылка на адрес Q в модуле A эквивалентна ссылке на адрес S + 10 в модуле B, а ссылка на байт T в модуле B эквивалентна ссылке на байт Q + 12 в модуле A (ввиду того, что при выравнивании адреса на границу слова между S + 19 и T образуется промежуток в два байта).

Команды ENTRY не нужны, так как имена P и R автоматически заносятся в ESD соответствующих модулей.

Рассмотрим теперь организацию программы динамической структуры. Модули, предназначенные для возможного включения в ее состав, хранятся в системной библиотеке в виде отдельных загрузочных модулей и загружаются в память динамически, по мере надобности, во время исполнения программы. Множество модулей, вовлекаемых в работу при конкретном исполнении динамической структуры, может изменяться в зависимости от входных данных решаемой задачи, наличия свободной памяти и других внешних условий. При этом обычно один и тот же участок оперативной памяти в разные моменты времени используется для размещения разных модулей, что позволяет существенно экономить место при исполнении больших программ.

Поскольку взаимное расположение модулей программы заранее не известно, между ними невозможно планировать связи, подобные приведенным в пунктах в) или г) в простой структуре. Поэтому обычно в динамической структуре модули выступают в качестве обращающихся друг к другу независимых подпрограмм. Начальный модуль программы загружается в память и получает управление с помощью оператора языка управления заданиями

```
//      EXEC      PGM=имя модуля
```

Загрузка следующих модулей и передача управления в их точки входа осуществляется системной макрокомандой LINK, входящей в состав начального и следующих модулей. С точки зрения внутренней логики программы функции LINK подобны функциям макрокоманды CALL: она формирует список адресов параметров, загружает адрес этого списка в регистр 1 и передает управление на вход вызываемого модуля, оставив адрес возврата в вызывающий модуль в регистре 14. Основные операнды макрокоманды LINK приведены на следующей схеме:

```
имя   LINK { EP=имя модуля    } [ ,PARAM=(список    ] [,VL=1]
           { EPLOC=адрес      }   [          пара-  ]
           {    имени модуля  }   [          метров)]
```

Первый (обязательный в команде) ключевой операнд указывает либо имя точки входа в модуль (нормально совпадающее с именем модуля), либо адрес поля из восьми байтов, содержащего

это имя. Второй и третий операнды аналогичны соответствующим операндам макрокоманды CALL. Например, если из модуля А нужно обратиться к подпрограмме, оформленной в виде загрузочного модуля В, передав ему адреса P1 и P2 двух параметров, причем адрес P2 нужно пометить как последний в списке, в соответствующем месте А (на той стадии, когда он еще был входным модулем) должна быть помещена макрокоманда

LINK EP = B, PARAM = (P1, P2), VL = 1

Действия макрокоманды LINK на самом деле много сложнее, чем действия CALL. Они поясняются схемой на рис. 29. Расширение LINK в нашем примере, кроме загрузки адресов списка параметров и возврата в А в регистры 1 и 14, осуществляет с помощью команды SVC переход в управляющую программу, сооб-

Рис. 29. Схема действия микрокоманды LINK

щая ей через регистр 15 имя В вызываемого модуля. Управляющая программа производит поиск работоспособной * копии модуля в оперативной памяти (где она могла сохраниться, например, после предыдущего обращения к ней). В случае неуспеха модуль отыскивается в библиотеке и загружается в свободный участок памяти. Затем управление через регистр 15 передается в точку входа, а в регистр 14 заносится адрес возвата в управляющую программу. Поэтому макрокоманда EXIT по окончании исполнения модуля В совершает переход вновь к управляющей программе. Последняя делает отметку об освобождении копии В и возвращает управление в вызывающий модуль А по оставленному ей макрокомандой LINK адресу.

Иногда подпрограмму, используемую в динамической структуре (например, при ее больших размерах), удобно оформлять в виде двух или более последовательно исполняемых модулей. Для этих целей можно воспользоваться системной макрокомандой XCTL (eXecute ConTrol), имеющей формат

[имя] XCTL { EP = имя модуля / EPLOC = адрес имени модуля }

* Работоспособность копии зависит от свойств модуля — повторной используемости и повторной входимости — рассматриваемых в следующем параграфе настоящей главы.

Способ ее применения для обращения из А к подпрограмме, оформленной в виде двух последовательно исполняемых загрузочных модулей В1 ии В2, показан на рис. 30. Схема не точна в том смысле, что все три перехода, показанные стрелками, осуществляются через посредство управляющей программы. Макрокоманда XCTL в модуле В1, занимающая место макрокоманды EXIT, делает отметку об освобождении исполнившейся копии В1, находит существующую или загружает в память новую копию В2 и передает управление на ее вход. Перед исполнением XCTL в регистре 13 должен быть восстановлен адрес области сохранения модуля А. Вход в модуль В2 отличается от стандартного

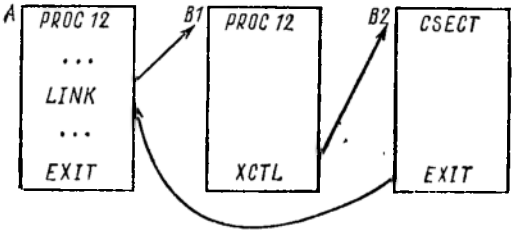


Рис. 30. Схема применения макрокоманды XCTL

в том, что там не нужно запоминать состояния регистров обратившегося модуля (они уже записаны в области сохранения А при входе в В1). Управляющая программа, через посредство которой XCTL вызывает модуль В2, может загрузить его на место, ранее занимаемое модулем В1.

Детальную информацию о форматах и действиях LINK и XCTL см. в [10] и [13].

Простая и динамическая структуры представляют крайние способы организации программы. Объединение всех модулей, требующихся для решения задачи, в один и однократная загрузка в память, запрашивая минимальное количество услуг от управляющей программы (и тем самым сокращая время исполнения), приводит неминуемо к большому расходу памяти для размещения программы. В противоположность этому в динамической структуре в каждый момент времени в памяти может храниться лишь цепочка подпрограмм, связанных еще не завершенными вызовами. Этот способ, предъявляя минимальные требования к размерам памяти, приводит к большому дополнительному расходу машинного времени, затрачиваемому управляющей программой на поиск работоспособных копий и перезагрузку модулей.

С целью экономии памяти и машинного времени в ОС ЕС ЭВМ предусмотрена третья форма, применяемая для организации больших программ, — структура с запланированным перекры-

тием (или оверлейная структура). Автор такой программы должен, исследовав все возможные пути связи между модулями, сгруппировать их в несколько сегментов: корневой сегмент, который будет постоянно находится в оперативной памяти, и перекрывающиеся сегменты, которые во время исполнения будут в определенной заранее последовательности и комбинациях замещать друг друга в оперативной памяти. Связь между модулями может осуществляться с помощью системных макрокоманд CALL. Задание редактору связей на выработку сегментов структуры, оформляемых в форме загрузочных модулей, состоит в указании в определенном порядке имен модулей с использованием управляющего предложения OVERLAY. Редактор связей добавляет к корневому сегменту служебную программу — оверлейный супервизор, обеспечивающий перезагрузку перекрывающихся сегментов, и строит таблицы, необходимые для его работы. Детальное описание сложной процедуры конструирования оверлейных программ содержится в руководствах [15, раздел 5.2] и [16].

§ 5.6. О типах загрузочных модулей

Каждый программный модуль в зависимости от своих внутренних свойств может быть невосстанавливающимся, восстанавливающимся (REUSable) или повторно входимым (REeNTerable). К первому из трех названных типов относятся такие модули, текст которых при исполнении претерпевает изменения, препятствующие повторному исполнению модуля с прежним результатом. Если к копии модуля, загруженной в память, можно последовательно обращаться много раз, модуль называется восстанавливающимся (или повторно используемым). Повторно входимым называется такой модуль, который кроме восстанавливаемости обладает еще одним свойством: каждое обращение к нему можно прервать на любой команде, а через некоторое время продолжить (восстановив предварительно состояния всех регистров процессора), независимо от всех других обращений к нему, которые могли иметь место во время прерывания. Тип модуля фиксируется редактором связей в соответствующем элементе оглавления библиотеки загрузочных модулей на основании характеристики REUS или RENT, указанной автором на стадии редактирования (по умолчанию модуль считается невосстанавливающимся). Информация о типе существенно используется управляющей программой при исполнении его копии в составе динамической структуры.

При написании программного модуля обычно стараются придать ему свойство восстанавливаемости. К таким модулям можно, например, обращаться из тела цикла в простой структуре. При наличии достаточных резервов памяти повторное обращение к ним по макрокоманде LINK в динамической структуре не

будет приводить каждый раз к перезагрузке модуля из библиотеки. Невосстанавливающиеся модули, переменные которых инициализируются лишь во время загрузки в память, могут оказаться полезными только в исключительных случаях.

Повторно входимые модули полезны в мультипрограммных операционных системах. Рассмотрим рис. 31. Здесь A и B — параллельно исполняемые программы. (В операционной системе

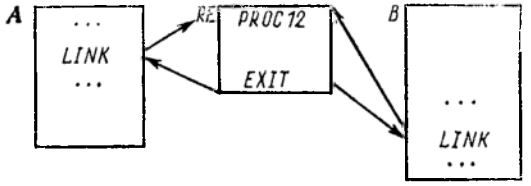


Рис. 31. Использование повторно впаденного модуля

MFT это могут быть шаги двух заданий, а в MVT — также параллельные задачи одного шага задания). Обе программы вызывают в качестве подпрограммы один и тот же модуль RE. Управляющая программа системы следит за эффективным использованием ресурсов. В то время, когда программа B ожидает данные, которые ей передает канал, процессор отдается в распоряжение программы A, и наоборот. Такие переключения могут происходить в заранее не предсказуемых местах. Может, например, оказаться, что процессор будет «отобран» у программы A в середине исполнения подпрограммы RE и передан B, которая, в свою очередь, обратится к тому же самому экземпляру RE. Если RE является обычным (хотя бы и восстанавливающимся) модулем, значения его переменных, хранивших информацию об обращении A, будут утрачены. Чтобы этого не произошло, необходимо размножать переменные RE при повторных входах, подобно тому, как это делается в рекурсивной подпрограмме. Но ограничения на структуру повторно входимого модуля, вообще говоря, тяжелее ограничений, накладываемых на рекурсивную подпрограмму. Для пояснения этого рассмотрим схему подпрограммы, приведенную на рис. 32, подобную рассмотренной ранее подпрограмме для игры «Ханойские башни».

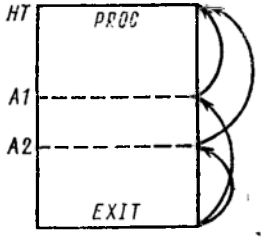


Рис. 32. Схема подпрограммы HT

Повторные обращения к HT возможны лишь в точках A1 и A2, отвечающих операторам HT (n — 1, p, 6 — p — q) и HT (n — 1, 6 — p — q, q). Если бы мы на любом из участков подпрограммы: от HT до A1, от A1 до A2, от A2 до EXIT пользовались какими-либо неразмножаемыми переменными (расположенными, например, между командами участка), которые хранили бы

нужные значения лишь во время исполнения команд одного участка, никакой потери информации при повторном входе не было бы, так как к моменту повторного входа в НТ значения этих переменных уже использованы и больше не нужны.

В повторно входимом модуле подобные вольности недопустимы, так как повторное обращение к нему может произойти после завершения любой его команды.

Для того чтобы ссылаться на переменные повторно входимого модуля с помощью ассемблерных имен, удобно пользоваться фиктивной программной секцией, построенной командой ассемблера DSECT. Пусть, например, в RE используются переменные P, Q и R, занимающие вместе поле в 800 байтов. Отведем для базисного адреса этих переменных регистр 11. Тогда структура модуля может быть такой:

```
RE      PROC        12, TM = NO
        GETMAIN     R, LV = 800        заказ памяти для
*                                      переменных
        LR          11, 1              загрузка базис-
*                                      ного адреса
        USING       V, 11              назначение базис-
*                                      ного регистра для
*                                      переменных
        ...                            команды модуля
        FREEMAIN    R, LV=800, A=(11)  освобождение па-
*                                      мяти при выходе
        EXIT
        ...                            постоянные мо-
*                                      дуля
V       DSECT
P       DS          ...                описания
Q       DS          ...                переменных
R       DS          ....               модуля
        END
```

Соответствие между обращениями к RE из разных программ и относящимися к ним выходами обеспечивается информацией в регистрах процессора, которую при прерываниях запоминает, а при продолжении обращения восстанавливает управляющая программа операционной системы.

Ассемблерные имена, описанные в фиктивной секции, нельзя использовать в адресных постоянных. Это накладывает ограничения на употребление P, Q, R в качестве операндов некоторых макрокоманд. Например, если мы хотим обратиться к подпрограмме S с параметром X, расположенным в фиктивной области, нельзя написать

```
        CALL        S, (X)
```

Нужно предварительно загрузить адрес X в какой-нибудь собственный регистр и указать последний в качестве операнда

```
        LA      2, X
        CALL    S, ((2))
```

Однако и последняя форма макрокоманды CALL не годится для повторно входимого модуля. Дело в том, что расширения обычных форм некоторых макрокоманд содержат не только машинные команды, но и переменные, что в рассматриваемом случае недопустимо. Например, расширение макрокоманды

```
        CALL    S, ((2))
```

выглядит так:

```
        CNOP    0, 4
        L       15, L2
        ST      2, L1       выгрузка адреса параметра
        BAL     1, L3
L1      DS      F           место для адреса параметра
L2      DC      V(S)
L3      BALR    14, 15
```

Мы видим, что содержимое слова L1 меняется при исполнении расширения.

Каждая макрокоманда, обладающая таким свойством, может быть разделена на две части: исполнительную, расширение которой не содержит никаких переменных, и описательную, содержащую только переменные, используемые командами исполнительной части или вызываемыми этими командами системными программами.

Например, если мы хотим в модуле RE обратиться к его внутренней подпрограмме S с параметрами P и Q, в месте вызова нужно поместить команды загрузки адресов и исполнительную форму CALL

```
        LA      2, P
        LA      3, Q
        CALL    S, ((2), (3)), MF=(E, LIST)
```

а в области V, увеличив ее размер по крайней мере на два машинных слова, написать описательную форму

```
LIST    CALL    ,(,), MF=L
```

или просто

```
LIST    DS      2F
```

так как список переменных состоит из двух машинных слов.

Подробнее об исполнительных и описательных формах системных макрокоманд см. в [13].

§ 5.7. О редактировании больших программ

Обычный путь составления достаточно больших программ состоит в расчленении алгоритма на процедуры и в написании и отладке этих процедур в виде программных модулей. Текст отлаживаемого модуля на языке ассемблера до завершения отладки хранится в библиотеке входных модулей и отлаживается с помощью небольших добавок, имитирующих обращения к нему и печатающих на АЦПУ отладочную информацию. После завершения отладки модуль обычно перерабатывается в загрузочную форму и помещается в библиотеку загрузочных модулей, откуда редактор связей может брать его для включения в простую или оверлейную структуру, или откуда он может загружаться в оперативную память для исполнения в составе динамической структуры.

Для облегчения процесса редактирования в системной библиотеке ОС ЕС ЭВМ, носящей имя SYS1.PROCLIB, хранятся описания стандартных процедур обращения к компилятору с языка ассемблера, редактору связей и управляющей программе. При употреблении этих процедур нужно указывать значения параметров компилятора и редактора и добавлять карты с описанием нужных наборов данных и управляющими предложениями редактора связей. Мы здесь не можем входить в многочисленные детали, изложенные в [15—17]. Ограничимся двумя примерами описания простых заданий на компиляцию и редактирование.

Пример 1. Входной модуль, хранящийся в библиотеке NAME1 входных модулей под именем NAME2, перевести в загрузочную форму и поместить в библиотеку NAME3 загрузочных модулей под именем NAME4. Для этой цели можно воспользоваться процедурой ASMFCL. Две последние буквы имени процедуры указывают названия вызываемых ею программ: компилятор (Compiler) и редактор связей (Linkage editor). Описание задания состоит из пяти карт:

```
//                  JOB      ...
//                  EXEC     ASMFCL,
//                    PARM.LKED='REUS, NCAL'
//ASM.SYSIN         DD       DSN=NAME1(NAME2),
//                           DISP=SHR
//LKED.SYSLMOD      DD       DSN=NAME3(NAME4),
//                           DISP=MOD
//
```

В обращении к процедуре указаны два параметра, управляющие работой редактора связей. Параметр REUS предписывает пометить изготавливаемый загрузочный модуль как вос-

станавливающийся. Параметр NCAL необходим в том случае, когда модуль предназначен для включения в простую структуру и содержит в своем ESD ссылки на другие модули.

Пример 2. Скомпилировать, отредактировать в простую структуру и выполнить программу, состоящую из двух модулей: главного, текст которого на языке ассемблера хранится на перфокартах, и вызываемого главным модуля NAME4, находящегося в библиотеке NAME3 загрузочных модулей. Описание задания содержит обращение к процедуре ASMFCLG, предусматривающей компиляцию, редактирование и выполнение программы:

```
//              JOB         ...
//              EXEC        ASMFCLG
                перфокарты с текстом главного модуля
//LKED.SYSIN    DD          *
                INCLUDE     LIB(NAME4)
//LKED.LIB      DD          DSN=NAME3, DISP=SHR
//GO.IN         DD          *
                перфокарты с входными данными программы
//GO.OUT        DD          SYSOUT=A
//
                Перфокарта с текстом
                INCLUDE     LIB(NAME4)
```

содержит управляющее предложение редактора связей, предписывающее присоединить к главному модулю загрузочный модуль NAME4 из библиотеки NAME3.

Глава 6

МАКРОПРОЦЕССОР АССЕМБЛЕРА ЕС ЭВМ

§ 6.1. Макроязык и макропроцессор

До сих пор мы рассматривали входную программу на языке ассемблера как последовательность предложений трех видов: машинных команд, команд ассемблера и макрокоманд, а компилятор с языка ассемблера — как некоторый программный автомат, который последовательно, в порядке их написания, анализирует предложения входной программы и также последовательно заменяет каждое предложение первого и третьего вида (и некоторые второго вида) определенными двоичными кодами, представляющими команды и постоянные машинной программы. При этом ничего не говорилось, по каким правилам компилятор обрабатывает макрокоманды и как автор программы может влиять на этот процесс.

На самом деле средства языка ассемблера много богаче, чем мы считали до сих пор. Их можно разделить на две части: базисный язык и макросредства. К базисному языку относятся машинные команды и все, рассмотренные до настоящего времени команды ассемблера: CSECT, DSECT, USING, END, DC, DS, ORG, PRINT, COM, ENTRY, EXTRN, DROP, CNOP, EQU, LTORG*.

К макроязыку относятся макрокоманды и следующие команды ассемблера: ACTR, AGO, AIF, GBLA, GBLB, GBLC, LCLA, LCLB, LCLC, MACRO, MEND, MEXIT, MNOTE, SETA, SETB, SETC, которые мы в отличие от команд базисного языка будем называть операторами. В свою очередь, в макроязыке можно различить две группы средств: макросы — операторы и конструкции, используемые для обработки макрокоманд, и операторы условной генерации, предназначающиеся как для обработки макрокоманд, так и для других целей.

---

* К базисным средствам относятся также команды ассемблера: CCW, COPY, CXD, DXD, EJECT, ICTL, ISEQ, OPSYN, PUNCH, REPRO, SPACE, START, TITLE.

Процесс преобразования входного модуля в объектный код состоит из двух последовательных стадий, выполняемых отдельными частями компилятора: макропроцессором (называемым также макрогенератором) и компилятором базисного языка. На первой стадии макропроцессор по указаниям, описанным автором программы средствами макроязыка, перерабатывает входной текст в текст, представляющий последовательность базисных команд. На второй стадии компилятор с базисного языка перерабатывает выходной текст макропроцессора в код объектного модуля. Вторая стадия принципиально проста. В основном она сводится к переводу постоянных в двоичную форму, к замене мнемонических обозначений машинных команд соответствующими двоичными числами, к вычислению смещений и к перестановке записей на поле операндов машинных команд в порядке, требуемом машинным форматом. На первой же стадии программа подвергается гораздо более глубокой переработке, требующей обычно нелинейного просмотра. Некоторые ее части при этом могут исключаться, а другие — многократно включаться с определенными изменениями в выходной текст на базисном языке.

Задача следующих разделов настоящей главы — познакомить читателя с возможностями макропроцессора и научить его использовать эти возможности для облегчения своего труда при написании программ.

§ 6.2. Простейшие макросы

Макросы — макроопределения и макрокоманды — являются средствами для реализации методом компиляции в языке ассемблера механизма процедур. Существо дела поясняет рис. 33. Его левая часть представляет входной текст макропроцессора, состоящий из двух частей. Первую часть этого текста — макроопределение — составляет описание некоторой процедуры P в терминах формальных операндов X и Y. Вторая часть — собственно программа — состоит из кусков 1, 2 и 3, содержащих только команды машины и ассемблера, и двух макрокоманд — операторов обращения к процедуре P с фактическими операндами соответственно A,B и C,D.

На правой части рис. 33 изображен результат работы макропроцессора, который, оставляя без изменения части 1, 2 и 3, заменяет каждую макрокоманду переработанным для соответствующего обращения текстом макроопределения.

Такой метод реализации процедур в отличие от рассмотренного в главе 5 метода интерпретации приводит, как правило, к более длинным программам. Но зато время исполнения такой программы процессором машины обычно меньше, так как в ней отсутствуют команды перехода к подпрограмме и обратно и не производится выбор адресов параметров из списка,

передаваемого подпрограмме через общий регистр 1. Применение макросов становится особенно выгодным для оптимизации машинной программы, если в них используются средства условной генерации (см. § 6.4 и 6.5). Однако в этом случае за эффективность машинной программы приходится платить значительным увеличением машинного времени, требующегося на работу макропроцессора.

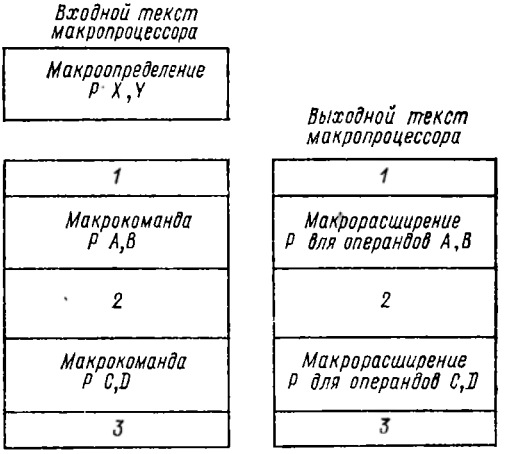


Рис. 33. Переработка входной программы макропроцессором

Познакомимся теперь со структурой макроопределения, изображенной на следующей схеме:

MACRO
Макропрототип
Последовательность модельных предложений
MEND

Первую строку каждого макроопределения составляет оператор

MACRO

Поля имени и операндов этого оператора должны быть заполнены пробелами.

Следующим предложением макроопределения является макропрототип, определяющий имя и формат макрокоманды. Он имеет четыре стандартных поля, разделяемых пробелами: имени, операции, операндов и комментария и обычно занимает позиции от 1 до 71 в строке бланка *. Поле операции макро-

* В общем случае макропрототип может иметь другой формат и занимать произвольное количество строк бланка (см. § 6.3).

прототипа содержит идентификатор длиной не более восьми символов, выбранный автором в качестве имени макрокоманды. Этот идентификатор должен отличаться от названий команд машины и ассемблера и всех остальных макрокоманд.

На поле операндов макропрототипа должны быть написаны отделенные друг от друга запятыми параметры макроопределения (играющие роль формальных операндов макрокоманды). Каждый параметр начинается специальным символом &, называемым амперсандом, за которым следует идентификатор, состоящий не более чем из семи букв или цифр.

Поле имени макропрототипа может быть либо пустым, либо содержать еще один параметр, который обычно служит для перенесения ассемблерного имени с поля имени макрокоманды на ее расширение.

Примером макропрототипа с одним параметром на поле имени и тремя на поле операндов может служить

```
&L      EXAMPLE1    &NAME, &NUM, &LOC
```

Модельные предложения являются прообразами будущих команд базисного языка, составляющих макрорасширение. Они также обычно имеют четыре поля и размещаются каждое в позициях от 1 до 71 строки бланка*.

Поле имени модельного предложения может быть пробелом, содержать ассемблерное имя, параметр, а также конкатенации (соединения) параметров с последовательностями букв и цифр и другими параметрами. Правило конкатенации состоит в следующем. Соединение параметра с параметром и присоединение последовательности символов к параметру слева от последнего производится непосредственным объединением их текстов. Если же мы хотим присоединить последовательность букв или цифр (или других символов в конкатенациях на поле операндов) к параметру справа, между ним и присоединенным текстом нужно поместить точку. Примерами конкатенаций могут служить

```
&NAME&LOC A&NAME &NAME.A1 &LOC.25&LOC
```

При присоединении к параметру справа текста, начинающегося с символа, отличного от буквы, цифры, точки или левой круглой скобки, соединительную точку употреблять не обязательно.

Поле операции модельного предложения может содержать название машинной команды или команды базисного ассемблера, отличной от END, ICTL, ISEQ, OPSYN, PRINT **. На нем

---

* Модельные предложения можно, используя правило продолжения (см. [5, раздел 3.1.2]), писать в трех последовательных строках. Они могут также представлять внутренние макрокоманды, занимающие несколько строк бланка (см. по этому поводу § 6.3).

** Поле операции может содержать также название макрокоманды, как другой, так и той, в макроопределении которой находится рассматриваемое предложение. Однако последнее имеет смысл лишь при использовании средств условной генерации.

можно также располагать конкатенации параметров с другими параметрами или с последовательностями букв (но с тем условием, чтобы при замене параметров их значениями на поле операции не возникло название макрокоманды, оператора или одной из пяти только что перечисленных команд базисного языка).

Поле операндов модельного предложения может содержать последовательности символов из алфавита ассемблера и их конкатенации с параметрами, составленные с таким расчетом, чтобы после замены параметров их значениями образовалась синтаксически правильная последовательность операндов соответствующей машинной команды, команды базисного ассемблера (или макрокоманды).

Поле комментария может содержать произвольный текст. Этот текст не обрабатывается макропроцессором. Он без всякого изменения (за исключением уменьшения количества ведущих пробелов) переносится в поле комментария порожденной команды.

Заключительным предложением каждого макроопределения служит оператор

MEND

поле операндов которого пусто, а поле имени содержит пробел или метку перехода макропроцессора (см. § 6.5).

Перейдем теперь к изучению структуры макрокоманды. Поле имени макрокоманды может быть занято ассемблерным именем или состоять из пробелов, а поле операции должно содержать название макрокоманды, совпадающее с идентификатором на поле имени макропрототипа в одном из макроопределений.

Количество операндов макрокоманды может быть большим. Поэтому формат ее поля операндов может отличаться от формата аналогичного поля у машинных команд и команд ассемблера. Мы подробно коснемся этого в следующем разделе, а пока ограничимся рассмотрением структуры одного операнда. Им может быть последовательность произвольных символов из алфавита машины в количестве не более 255, удовлетворяющая некоторым синтаксическим ограничениям. Перечислим кратко эти ограничения (см. [5, раздел 4.3.1] или [18, раздел 5.2]).

Апострофы могут использоваться для образования строк — последовательностей символов, ограниченных справа и слева апострофами. Ограничивающие строку апострофы называются парными. Строки могут содержать внутри себя другие строки. Круглые скобки, знаки равенства, запятые и пробелы в составе строк теряют свое служебное значение. Для представления апострофа внутри строки используются два соседних знака апострофа. Они не считаются парными. Например, в записи

'A''B'C'D'

парными являются лишь первый и четвертый, пятый и шестой апострофы. Одиночные апострофы вне строк могут также упот-

ребляться для указания характеристики длины ассемблерного имени. Для этого апостроф должен располагаться между буквой L и ассемблерным именем, причем перед буквой L должен находиться специальный символ, отличный от амперсанда. Например:

A + L'NAME

Количество круглых скобок в операнде, не входящих в состав строк, должно быть четным. При этом n-я левая скобка должна располагаться левее n-й правой.

Знак равенства, если он не входит в состав строки или не содержится между левой и правой скобкой одного уровня, может быть только первым символом операнда. Примеры допустимых вхождений знака равенства:

```
   =F'32'
'C=0'
E(F=G)
```

Запятая, не входящая в состав строки и не заключенная между парными круглыми скобками, используется только в качестве разделителя операндов.

Пробел вне строки служит признаком конца поля операндов.

Одиночный знак амперсанда может использоваться лишь в составе обозначения параметра (а также системной переменной или переменной макропроцессора). Для представления амперсанда как символа алфавита используются два соседних знака:

&&125&&&&

Количество операндов макрокоманды должно, вообще говоря, совпадать с количеством параметров в макропрототипе. Разрешается также опускать некоторые операнды, оставляя без пробела ограничивающие их запятые. Например, прототипу EXAMPLE1, описанному на с. 151, могут соответствовать такие макрокоманды:

```
A       EXAMPLE1    ABC, ,F
        EXAMPLE1    DEF, 25,
```

Запятые, расположенные справа от последнего непустого операнда, выписывать необязательно. Поэтому вторая макрокоманда эквивалентна такой:

```
        EXAMPLE1    DEF, 25
```

Нам осталось теперь сформулировать правило, по которому макропроцессор вырабатывает заменяющее макрокоманду расширение. Правило состоит из двух шагов. На первом шаге каждому параметру на поле операндов макропрототипа присваивается в качестве значения текст соответствующего ему в порядке расположения (включая и опущенные) операнда.

Параметру, которому отвечает опущенный операнд, присваивается пустое значение. Параметр, расположенный на поле имени прототипа, приобретает в качестве значения ассемблерное имя, расположенное на поле имени макрокоманды. Если имя отсутствует, параметр получает пустое значение.

На втором шаге все параметры, входящие в состав трех первых полей модельных предложений, замещаются присвоенными им в качестве значений текстами. При этом точки, употребленные при конкатенации параметров с расположенными справа от них текстами, опускаются.

Для иллюстрации изложенного рассмотрим два примера.

Пример 1. Макроопределение MOVE

```
        MACRO
&L      MOVE &TO,&FROM,&F
&L      L&F 0,&FROM
        ST&F 0,&TO
        MEND
```

содержит два модельных предложения и порождает расширение, состоящее из двух машинных команд. Макрокоманде

```
A       MOVE    P, Q, H
```

соответствует расширение

```
A       LH      0, Q
        STH     0, P
```

Обращение к MOVE по макрокоманде

```
        MOVE    F1, F2
```

породит расширение

```
        L       0, F2
        ST      0, F1
```

так как параметр &F в этом случае получит пустое значение.

Пример 2. Более сложное макроопределение

```
        MACRO
&L      SUM     &S, &BEG, &NUM
&L      SR      1, 1                    начальный индекс равен 0
*
        LA      14, 4                   шаг изменения индекса равен 4
*
        LA      15, 4*(&NUM-1)          конечное значение индекса
*
        SR      0, 0                    место для суммы
```

```
         A        0, &BEG.(1)        добавление очередного
*                                    слагаемого
         BXLE     1, 14, *—4         переход к предыдущей
*                                    команде
         ST       0, &S
         MEND
```

служит для образования в машинном слове &S суммы последовательности двоичных чисел, расположенных в &NUM соседних машинных словах, начиная со слова с адресом &BEG. В этом макроопределении в третьем и пятом модельных предложениях используются конкатенации параметров с прилегающими справа текстами

&NUM — 1

и

&BEG.(1)

В первом случае дополнительная точка опущена, так как присоединяемый текст начинается с символа минус, отличного от буквы, цифры, точки или левой скобки. Во втором случае точку опустить нельзя, так как тогда текст модельного предложения, оставаясь синтаксически правильным, приобретет другой смысл (см. § 6.3 настоящей главы).

Макрокоманде

```
LABEL    SUM      A, B, 10
```

соответствует следующее макрорасширение:

```
LABEL    SR       1, 1               начальный индекс равен 0
*
         LA       14, 4              шаг изменения индекса равен 4
*
         LA       15, 4*(10—1)       конечное значение индекса
*
         SR       0, 0               место для суммы
         A        0, B(1)            добавление очередного
*                                    слагаемого
         BXLE     1, 14, *—4         переход к предыдущей
*                                    команде
         ST       0, A
```

Заметим, что в рассмотренном случае макропроцессор не вычисляет значения выражения на поле второго операнда третьей команды макрорасширения. Однако, как мы увидим, этого можно добиться с помощью средств условной генерации.

В заключение раздела опишем правила для включения макроопределений во входную программу.

По характеру использования их макропроцессором все

макроопределения можно разделить на две категории: макроопределения пользователя и системные макроопределения. Автор программы сам решает вопрос, к какой из них отнести то или иное определение.

Все определения пользователя помещаются во входной программе перед первой командой CSECT (или перед макрокомандой PROC). Обрабатывая программу, макропроцессор сначала производит синтаксический анализ всех макроопределений пользователя и печатает их тексты в начале листинга, указывая обнаруженные им синтаксические ошибки. Затем он, даже при наличии ошибок, делает попытку заменить соответствующие макрокоманды их расширениями. Если в программе не было команды ассемблера

```
PRINT    NOGEN
```

тексты расширений на базисном языке включаются в состав листинга. При этом повторно появляются сообщения о синтаксических ошибках. Но так как они дополняются сообщениями об ошибках, обнаруженных компилятором с базисного языка в тексте расширения, найти первоначальную ошибку по этим вторичным указаниям довольно трудно. Поэтому новые определения, вплоть до завершения отладки, следует относить к категории определений пользователя.

К системным макроопределениям относятся такие, тексты которых храняться в виде отдельных разделов в системной библиотеке с именем SYS1.MACLIB. Текст отлаженного определения можно либо записать в эту библиотеку, либо, если соответствующая макрокоманда нужна лишь отдельным пользователям, хранить его в личной библиотеке, присоединяя последнюю к системной лишь во время выполнения того пункта задания, в котором происходит обращение к ассемблеру. Это можно сделать, добавив в соответствующее место описания задания управляющие карты

```
//ASM.SYSLIB    DD    DSN=SYS1.MACLIB,
//                    DISP=SHR
//              DD    DSN=LIBNAME, DISP=SHR
```

где идентификатор LIBNAME замещает имя той личной библиотеки, в которой хранятся нужные макроопределения. При этом имена разделов LIBNAME, составляющих тексты макроопределений, должны совпадать с именами соответствующих макрокоманд.

Тексты системных макроопределений не включаются в состав листинга и их предварительный синтаксический анализ не производится. Об ошибках в них автор может узнать только по сообщениям, сопровождающим макрорасширение. Если макроопределение с одним и тем же именем входит в состав обеих категорий, макропроцессор использует текст определения пользователя.

§ 6.3. Дополнительные средства макросов

В § 6.2 мы, описывая структуру макроопределения и макрокоманды, говорили только о позиционных параметрах и соответствующих им позиционных операндах макрокоманды. Наряду с ними при написании макроопределения можно пользоваться ключевыми параметрами, которые могут иметь априорные (присвоенные по умолчанию) значения. Каждый ключевой параметр макропрототипа состоит из знака амперсанда, идентификатора длиной не более семи символов (ключа), знака равенства и следующего за ним априорного значения, которое, в частности, может быть пустым. Список параметров макропрототипа может состоять либо только из позиционных параметров, либо только из ключевых, либо содержать как те, так и другие. В последнем случае ключевые параметры должны располагаться после позиционных. Примером макропрототипа со смешанным списком может служить

```
&L EXAMPLE2 &P1, &P2, &REG = 0, &F =, &NAME = IN
```

который кроме параметра &L на поле имени имеет два позиционных &P1 и &P2 и три ключевых: &REG, &F и &NAME, по умолчанию получающих значения соответственно 0, пусто и IN.

Ключевой операнд макрокоманды состоит из идентификатора ключевого параметра и знака равенства, за которым следует текст, составляющий значение параметра. Ключевой операнд допустим лишь тогда, когда макропрототип содержит соответствующий ключевой параметр. Если в макрокоманде есть операнды обоих типов, ключевые должны располагаться после списка всех позиционных. Взаимный порядок ключевых операндов может отличаться от порядка соответствующих параметров в прототипе, а количество может быть меньше количества ключевых параметров. Например, обращение к EXAMPLE2 может выглядеть так

```
EXAMPLE2      A, NAME = OUT
```

В модельных предложениях макроопределения ключевые параметры применяются точно так же, как и позиционные, но значения им присваиваются по-другому. Те параметры, ключи которых присутствуют в списке операндов, приобретают значения, указанные в соответствующих операндах. Все остальные ключевые операнды сохраняют свои априорные значения. Например, при образовании расширения макрокоманды

```
EXAMPLE2      A, NAME = OUT
```

параметры &L, &P2, &F получают пустые значения, &REG — значение 0, &NAME — значение OUT, а &P1 — значение A.

Употребление ключевых параметров и операндов удобно в макрокомандах с большим количеством операндов, значительная часть которых при конкретных обращениях должна получать

стандартные значения. Например, если мы заменим макропрототип определения SUM в примере 2 на такой

```
&L      SUM     &S, &BEG, &NUM, &R0=0, &R1=1,      *
                &R2=14
```

а в модельных предложениях вместо номеров регистров 0, 1, 14 и 15 напишем соответственно выражения &R0, &R1, &R2 и &R2+1, мы получим более гибкую макрокоманду, в расширении которой вместо стандартных общих регистров можно использовать любые, указываемые при желании дополнительными ключевыми операндами.

Так как поле операндов макропрототипа может содержать произвольное количество позиционных и ключевых параметров, его разрешается располагать на нескольких соседних строках бланка, используя два правила переноса. Первое состоит в том, что, заполнив очередным символом поля операндов позицию 71 бланка, заносят в позицию 72 любой символ, отличный от пробела (не входящий в состав записи на поле операндов) и продолжают запись с позиции 16 следующей строки. Согласно второму правилу, мы можем оборвать запись в очередной строке на запятой, поставленной за очередным операндом, поместить после нее по крайней мере один пробел, в позицию 72 вписать символ, отличный от пробела, и начать запись следующих операндов с позиции 16 следующей строки. Между упомянутым выше пробелом и позицией 72 можно помещать произвольный текст, служащий комментарием.

Оба правила можно применять в любом порядке неограниченное количество раз. Концом растянутого таким образом поля операндов считается первый пробел в строке бланка, такой, что пробел содержит также либо позиция 72 той же строки, либо позиция 16 следующей строки.

Точно такие же правила переноса можно применять при написании макрокоманды. При этом никакого согласования в расположении операндов по строкам в макрокоманде и соответствующем ей прототипе соблюдать не нужно.

Комментарий в составе макроопределения может располагаться не только между параметрами макропрототипа и на полях комментария макропрототипа и модельных предложений, но также и в специальных модельных предложениях, содержащих символ * в позиции 1 или пару символов .* в позициях соответственно 1 и 2 строки бланка. В макрорасширение переносятся только комментарии, расположенные на поле комментария модельных предложений и в предложениях, начинающихся символом * в первой позиции. Никакой обработки их не производится, и параметры, включенные в их состав, значениями не заменяются. Предложения, начинающиеся с комбинации точки и звездочки, не включаются в листинг макрорас-

ширения. В них обычно сообщаются сведения, нужные лицам, составляющим макроопределения и бесполезные для программистов, только использующих макрокоманды.

Некоторые операнды макрокоманды иногда удобно объединять в группы, называемые подсписками. Подсписок состоит из нескольких, разделенных запятыми, операндов, заключенных в скобки. Он рассматривается как единый операнд, ключевой или позиционный, сопоставляемый одному параметру макропрототипа. В подсписки обычно объединяют однородные по свойствам операнды. Употребляя средства условной генерации, можно пользоваться подсписками переменной длины (см. § 6.3).

Рассмотрим пример, иллюстрирующий действия с подсписками.

Пример 3. Макроопределение UNION предназначено для объединения в одном поле &S трех текстов, указанных тремя ассемблерными именами их первых байтов и тремя числами, равными длинам текстов.

```
      MACRO
&L    UNION   &NSBL, параметр-подсписок с тремя именами *
                &LSBL, параметр-подсписок с тремя длинами *
                &S поле для результата
&L    MVC     &S.(&LSBL(1)), &NSBL(1)
.* перенос первого текста длины &LSBL(1) в начало поля &S
      MVC     &S+&LSBL(1)(&LSBL(2)), &NSBL(2)
.* перенос второго текста на место &S+&LSBL(1)
      MVC     &S+&LSBL(1)+&LSBL(2)(&LSBL(3)), *
                &NSBL(3)
.* перенос третьего текста на место &S+&LSBL(1)+&LSBL(2)
      MEND
```

В этом макроопределении поле операндов прототипа, содержащее три параметра, занимает три строки бланка. Выражения &LSBL(n) и &NSBL(n) в модельных командах являются не конкатенациями параметров с текстом (n), а представляют синтаксические конструкции, заменяемые n-м подпараметром из соответствующего подсписка. Целое положительное число n указывает порядковый номер параметра. Если n больше количества элементов подсписка, конструкция заменяется пустым значением.

Пусть начала объединяемых текстов указаны именами P, Q и R, а их длины соответственно равны 5, 3 и 9. Тогда макрокоманда

```
      UNION   (P, Q, R), (5, 3, 9), F
```

заменится следующим расширением:

```
MVC     F(5), P
MVC     F+5(3), Q
MVC     F+5+3(9), R
```

Для обращения из модельных предложений к позиционным параметрам можно пользоваться системной переменной

&SYSLIST

Именно, конструкция

&SYSLIST(n)

эквивалентна наименованию n-го позиционного параметра, а конструкция

&SYSLIST(n, m)

эквивалентна ссылке на m-й подоперанд n-го параметра, рассматриваемого как подсписок. Здесь n и m означают целые положительные числа (или арифметические выражения, имеющие соответствующие значения — см. § 6.4). Если операндов или подоперандов с соответствующими номерами в макрокоманде нет, конструкция заменяется пустым значением.

Например, в макроопределении UNION первую модельную команду можно написать в такой форме

```
&L      MVC             &SYSLIST(3) (&SYSLIST(2, 1)), *
                        &SYSLIST(1, 1)
```

Если бы мы во всех модельных предложениях UNION выразили все ссылки на &NSBL, &LSBL и &S через &SYSLIST, можно было бы в макропрототипе UNION опустить все позиционные параметры, написав его в виде

```
&L      UNION
```

Упомянем еще одно средство макропроцессора, полезное в макроопределениях, порождающих команды перехода (см. пример 2). Адрес перехода обычно указывается ассемблерным именем, помечающим ту команду программы, переход к которой нужно совершить. Но внутри макроопределения помечать модельные предложения обычным именем опасно: если мы внутри входной программы более одного раза сошлемся на такое макроопределение, один и тот же идентификатор более одного раза появится на поле имени и компилятор с базисного языка отметит ошибку. Для избежания этого в ассортименте макропроцессора имеется системная переменная со стандартным обозначением &SYSNDX. Эта переменная может употребляться в модельных предложениях точно на таких же правах, как параметры макропрототипа. Она автоматически заменяется четырехзначным десятичным числом. Перед началом обаботки это

число равно 0000 и увеличивается на 1 перед входом в каждую новую макрокоманду. С помощью &SYSNDX можно генерировать различные ассемблерные имена. Для этого достаточно присоединить к ней слева какой-нибудь идентификатор длиной не более четырех символов. Например, в макроопределении SUM примера 2 можно было бы второе и третье снизу модельные предложения заменить такими:

```
L&SYSNDX    A     0,&BEG.(1)
            BXLE  1,14,L&SYSNDX
```

Тогда в первом макрорасширении конкатенация L&SYSNDX заменится идентификатором L0001, во втором — L0002 и т. д., и мы избежим повторного описания одного и того же имени.

В качестве модельных предложений можно употреблять и макрокоманды. Такие макрокоманды называются внутренними, в отличие от внешних макрокоманд, расположенных вне какого-либо макроопределения. Расширения внутренних макрокоманд образуются по тем же правилам и включаются в расширения внешних макрокоманд. Операнды внутренних макрокоманд могут содержать в своем составе параметры макропрототипа (а также и другие переменные макропроцессора), заменяемые соответствующими значениями перед началом обработки внутренней макрокоманды. Переменная &SYSNDX, увеличивающаяся на один перед началом обработки внутренней макрокоманды, получает прежнее значение по выходе из нее, которое используется при продолжении обработки модельных предложений внешнего макроопределения. На внутренние макрокоманды накладывается ограничение: элемент подсписка внешней макрокоманды, сам являющийся подсписком, не может быть подсписком во внутренней макрокоманде.

§ 6.4. Условная генерация

Мы уже упоминали о том, что макропроцессор, обрабатывая входной текст, может просматривать его нелинейно. Приведем простой пример, поясняющий эти функции.

Пример 4. Макроопределение ADD предназначено для образования в указанном общем регистре суммы нескольких двоичных длинных слагаемых, количество которых может быть различным при различных обращениях к ADD.

```
        MACRO
&L      ADD     &S, &R          &S — подсписок, содер-
*                                   жащий неизвест-
*                                   ное количество
*                                   слагаемых
*                               &R — номер общего ре-
*                                   гистра
```

```
         LCLA    &A                    описание перемен-
*                                      ной типа A
&A       SETA    1                     присваивание &A
*                                      значения 1
&L       L       &R, &S(1)             модельная ко-
*                                      манда
.LOOP    AIF     (&A EQ N'&S).END      оператор услов-
*                                      ного перехода
&A       SETA    &A+1                  увеличение значе-
*                                      ния &A на 1
         A       &R, &S(&A)            модельная ко-
*                                      манда
         AGO     .LOOP                 оператор безу-
*                                      словного пере-
*                                      хода
.END     MEND
```

Макропрототип ADD содержит два позиционных параметра: &S, заменяемый подсписком адресов слагаемых, и &R — номер регистра для размещения суммы.

Следующая строка макроопределения — оператор LCLA — составляет описание переменной &A, которая во время работы макропроцессора может хранить целые значения. Оператор SETA присваивает &A значение 1. За ним следует модельная команда L, предназначенная для загрузки в &R первого слагаемого.

Оператор AIF является оператором условного перехода макропроцессора. Его поле операндов содержит отношение

(&A EQ N'&S)

и метку перехода макропроцессора

.END

помечающую тот оператор, к выполнению которого следует перейти, если отношение имеет значение «да». В случае «нет» следует выполнять оператор, записанный непосредственно после AIF. Правая компонента отношения

N'&S

является синтаксической конструкцией, определяющей количество элементов в подсписке &S.

Оператор SETA, расположенный после AIF, при каждом исполнении увеличивает значение &A на 1. За ним следует модельная команда A, добавляющая в регистр &R очередное слагаемое (напомним, что конструкция &S(&A) замещается элементом подсписка &S, имеющим порядковый номер &A).

Оператор безусловного перехода

```
AGO      .LOOP
```

предписывает макропроцессору вернуться к исполнению оператора AIF.

Рассмотрим действия макропроцессора при обработке макрокоманды

```
ADD     (A, B, C), 2
```

В результате исполнения первых трех операторов рабочая переменная &A получает значение 1, а в макрорасширение поступает команда

```
L       2, A
```

Так как значения &A и N'&S при выполнении оператора AIF не совпадают, макропроцессор переходит к следующему оператору и увеличивает значение &A до 2. Затем — к модельной команде, которая порождает вторую команду расширения

```
A       2, B
```

Затем, вследствие безусловного перехода, второй раз исполняется оператор AIF. Так как и в этот раз &A не равно N'&S, значение &A вновь увеличивается на 1 и в расширение поступает команда

```
A       2, C
```

При третьем исполнении AIF отношение имеет значение «да». Поэтому макропроцессор совершает переход к оператору

```
MEND
```

прекращающему обработку макрокоманды. Полученное расширение (с точностью до комментария) есть

```
L       2, A
A       2, B
A       2, C
```

Средства условной генерации можно использовать и вне макроопределений.

Пример 5. Предположим, что при включении входного модуля A в различные программы простой структуры нам нужно изменять имя некоторой подпрограммы, макрокоманда обращения к которой встречается в тексте модуля несколько раз. Для упрощения процедуры замены мы можем придать модулю следующий вид.

```
        LCLC    &N              описание переменной типа C
*
A       PROC    12, RENT=NO     начало модуля A
&N      SETC    'ABC'           заменяемый оператор
        ...
```

```
-       CALL    &N, (P, Q)          модельная макро-
*                                   команда
        ...
        CALL    &N, (M, N)          модельная макро-
*                                   команда
        ...
        CALL    &N, (B, C)          модельная макро-
*                                   команда
        ...
        END
```

Мы добавили к тексту модуля два оператора условной генерации. Первый служит описанием переменной &N макропроцессора, которая может иметь значением последовательность символов. Второй оператор присваивает этой переменной в качестве значения идентификатор ABC. После обработки модуля макропроцессором во всех расширениях системной макрокоманды CALL в качестве имени подпрограммы будет фигурировать идентификатор ABC.

Если наш модуль хранится на перфокартах, для изменения имени подпрограммы нам достаточно заменить в нем третью по порядку карту.

§ 6.5. Средства условной генерации

Основными средствами условной генерации служат переменные макропроцессора типов A, B и C (называемые в некоторых руководствах переменными параметрами), операторы присваивания значений этим переменным и операторы перехода макропроцессора. Познакомимся подробнее с синтаксисом и семантикой этих понятий.

6.5.1. Локальные переменные типа A и арифметические выражения. Переменная типа A может принимать целые значения из диапазона от -2^{-31} до $2^{31} - 1$. Областью действия локальной переменной типа A может быть либо одно макроопределение, либо собственно программа. Для описания переменных предназначен оператор LCLA (LoCaL Arithmetic), имеющий формат:

пробелы LCLA список переменных комментарий

Имя переменной состоит из знака амперсанда, за которым следует идентификатор длиной не более семи букв или цифр. В одном операторе можно описать несколько переменных. Все они при этом автоматически получают начальное значение 0. Для определения локальных переменных собственно программы нужно поместить оператор (или операторы) LCLA перед первой командой CSECT программы. Описание внутренних переменных макроопределения производится операторами, расположенными непосредственно после макропрототипа. Одно и то же

имя в разных областях означает разные переменные. Имена внутренних переменных макроопределения должны быть отличными от имен параметров макропрототипа.

Оператор присваивания нового значения переменной типа A имеет формат:

переменная SETA арифметическое выражение комментарий

Он может располагаться в любом месте области действия переменной. Арифметическое выражение состоит из одного терма или из нескольких термов, соединенных знаками арифметических операций

$$+ \quad - \quad * \quad /$$

Для изменения обычного порядка действий можно использовать круглые скобки (но не более чем шести уровней). Значение выражения вычисляется из значений термов по правилам арифметики целых чисел. Результатом деления считается целая часть частного. Значения выражения и всех его промежуточных результатов не должны выходить за указанные выше пределы.

Перечислим конструкции, которые могут быть термами арифметического выражения.

В качестве терма можно употреблять самоопределенный терм — десятичное, двоичное, 16-ричное или символьное изображение целого числа:

25 B'10011' X'AB7' C'A'

Термом может быть переменная макропроцессора типа A или B (см. п. 6.5.3).

Переменная типа C может быть термом арифметического выражения лишь в тех случаях, когда ее значение не содержит символов, отличных от десятичных цифр, т. е. является изображением десятичного целого без знака, которое и считается значением терма.

В арифметическом выражении, расположенном внутри макроопределения, термом может быть параметр макропрототипа, но лишь в тех случаях, когда он замещается самоопределенным термом.

Термом арифметического выражения вне макроопределения может быть ссылка на характеристику длины ассемблерного имени, описанного в собственно программе, а в арифметическом выражении внутри макроопределения — ссылка на характеристику длины параметра макроопределения, замещаемого ассемблерным именем. Примеры таких термов:

L'NAME L'&P L'&S(2) L'&SYSLIST(2, 3)

Правила определения характеристики длины имени состоят в следующем. Если имя помечает машинную команду, характеристика длины равна длине команды в байтах. Если имя описано в команде ассемблера DS или DC, характеристика равна

длине соответствующей первой постоянной или переменной. Если имя описано в команде CSECT или DSECT, характеристика равна 1. Если имя определено в команде EQU, то характеристика равна 1 при абсолютном выражении на поле операндов, а в противном случае определяется по характеристике первого переместимого терма на поле операндов.

Значение ссылки на характеристику длины в арифметическом выражении не определено, если соответствующее имя описано внутри какого-либо макроопределения, описано в команде DS или DC, содержащей на поле модификатора длины выражение, отличное от самоопределенного терма.

В арифметическом выражении внутри макроопределений в качестве термов можно пользоваться характеристикой количества операндов в подсписке

N'&S

и характеристикой количества символов в операнде

K'&P

Значение N'&S определяется следующим образом. Если соответствующий операнд опущен, N'&S равно 0. Если &S соответствует одиночный операнд, N'&S равно 1. Если &S соответствует подсписок, то N'&S равно количеству операндов в подсписке, считая и опущенные операнды, если присутствуют ограничивающие их запятые.

Пусть, например, второй позиционный операнд макрокоманды есть:

(A, (B, C,),,)

Тогда N'&SYSLIST(2) имеет значение 4, N'&SYSLIST(2, 2) — значение 3, N'&SYSLIST(2, 1) — значение 1, а N'&SYSLIST (2, 3) — значение 0.

Терм N'&SYSLIST определяет количество позиционных операндов в макрокоманде.

Значение терма K'&P равно количеству символов, составляющих соответствующий операнд. В качестве примера в левой колонке следующей таблицы приведены конструкции, которые могут быть операндами макрокоманды, а в правой колонке — соответствующие значения характеристик K:

ABCD	4
(A, B, C, D)	9
2(10, 12)	8
'A''B'	6
''	3

Кроме перечисленных термов в арифметическом выражении можно также употреблять термы S'&P и I'&P (внутри макроопределений) или S'NAME и I'NAME (вне макроопределе-

ний), имеющие значения соответственно характеристики масштаба и характеристики целой части, присваиваемые ассемблерным именам, указывающим на числовые постоянные некоторых типов (см. по этому поводу [18, гл. 8]).

Арифметическое выражение, синтаксис и семантику которого мы только что описали, используется и в других конструкциях входного языка макропроцессора, отличных от правой части оператора SETA. На всем протяжении главы 5 под термином «арифметическое выражение» мы будем подразумевать именно такие выражения.

Переменные типа A могут фигурировать в следующих конструкциях: в левой части операторов SETA, в арифметических выражениях, в символьных выражениях (см. § 5.2) и в модельных предложениях как внутри, так и вне макроопределений. В модельных предложениях эти переменные можно использовать в конкатенациях с текстами, параметрами и другими переменными для формирования частей поля имени и операндов. При преобразовании модельного предложения в команду базисного языка переменная типа A заменяется десятичным изображением соответствующего значения без знака и ведущих нулей. Нулевое значение заменяется символом 0.

Например, если в области определения переменных &A1 и &A2 во входной программе расположены два оператора SETA и модельное предложение:

```
&A1     SETA    4
&A2     SETA    &A1 — 10
L&A2    L       &A1, A(&A2 + 2)
```

последнее породит машинную команду

```
L6      L       4, A(6 + 2)
```

6.5.2. Локальные переменные типа C и символьные выражения. Локальная переменная макропроцессора типа C принимает в качестве значения последовательность любых символов алфавита длиной не более 8. Ее значение может быть пустым (т. е. не содержать ни одного символа). Описание переменных типа C производится оператором LCLC (LoCaL Character), одновременно присваивающим всем определяемым переменным пустые значения. Обозначения переменных, формат оператора, области действия переменных и расположение оператора в программе полностью аналогичны только что рассмотренному случаю переменных типа A.

Значение переменной типа C может изменяться расположенным в ее области действия оператором SETC, имеющим формат

переменная SETC символьное выражение комментарий

Действие его заключается в том, что переменная, названная в левой части, получает новое значение, определяемое правой частью. В том случае, когда новое значение содержит более восьми символов, переменной присваиваются левые восемь.

Перейдем к описанию структуры символьных выражений.

а) Символьное выражение может быть ссылкой на характеристику типа, состоящей из буквы T, апострофа и следующего за ним ассемблерного имени (вне макроопределения) или (внутри макроопределения) обозначения параметра.

Каждое ассемблерное имя, описанное в собственно программе, относится макропроцессором к одному из 19 типов, обозначаемых определенными буквами. Например, имя, указывающее на адресную постоянную типа A, имеет тип A, имя двоичной постоянной имеет тип B, имя символьной постоянной — тип C, имя короткого двоичного числа с плавающей точкой — тип E, и т. д. (см. таблицу характеристик типов в [5, раздел 4.5.2] или [18, раздел 8.2]). Буква, обозначающая тип, и является значением ссылки на характеристику имени или параметра, заменяемого ассемблерным именем. Значение ссылки на характеристику типа параметра, который заменяется не ассемблерным именем, есть

N — в случае самоопределенного терма,

O — в случае опущенного операнда,

U — в остальных случаях*.

Примеры операторов SETC с характеристиками типа:

```
&C1     SETC    T'NAME
&C2     SETC    T'&SYSLIST(2)
```

Если операнд &SYSLIST(2) является подсписком, характеристика определяется по первому его элементу.

б) Символьное выражение может быть строкой — последовательностью произвольных символов алфавита в количестве до 255, ограниченной справа и слева парными апострофами, которые не входят в значение выражения. В следующих ниже примерах значения, присваиваемые переменным левой части, написаны на полях комментария соответствующих операторов.

```
&C1     SETC    'ABCD EFGH'     ABCD EFG
&C2     SETC    'L''NAME'       L'NAME
&C3     SETC    '&&BCD'         &&BCD
&C4     SETC    ''              пустое значение
```

в) Символьным выражением может быть заключенное в апострофы обозначение переменной макропроцессора типа C, A или B или (внутри макроопределения) заключенное в апострофы

* Тип U (Undefined) получают также имена, помечающие команды EQU и LTORG, и команды DS и DC, имеющие в качестве модификаторов выражения, отличные от самоопределенных термов.

обозначение параметра. Значение этого выражения есть значение соответствующей переменной или операнд макрокоманды. В случае переменной типа B значение — символ нуля или символ единицы, а в случае переменной типа A — изображение ее значения в виде десятичного целого без знака и ведущих нулей. Символ апострофа в составе значения представляется одним апострофом, а символ амперсанда — двумя символами. Например

```
&C1     SETC    '''A=B'''       'A=B'
&C2     SETC    '&C1'           'A=B'
&C3     SETC    '&&'            &&
&C4     SETC    '&C3'           &&
```

Если параметру &P1 отвечает операнд 'X=Y', а параметру &P2 — операнд (A, B, C, D), то &C5 и &C6 получат следующие значения:

```
&C5     SETC    '&P1'           'X=Y'
&C6     SETC    '&P2'           (A, B, C, D
```

г) Символьное выражение может быть также подстрокой, т. е. выражением вида б) или в), вслед за которым в круглых скобках написаны два арифметических выражения, разделенные запятой. Первое выражение указывает порядковый номер начального символа, а второе — длину последовательности символов, выделяемой в качестве значения подстроки.

Примеры операторов SETC с подстроками

```
&C1     SETC    'ABCD''EFG'(3, 4)     CD'E
&C2     SETC    '&C1'(2, 5)           D'E
```

Если второе арифметическое выражение больше количества символов справа от указанного первым, значение подстроки автоматически укорачивается. Если же первое выражение указывает на несуществующий символ, макропроцессор отмечает ошибку и оператор не выполняется.

д) Символьным выражением может быть конкатенация выражений видов б), в) и г) в любом порядке и количестве. Для соединения выражений используется дополнительная точка. Точку между правой скобкой и следующим апострофом разрешается опускать. Значением конкатенации является последовательность символов, полученная объединением значений всех составляющих символьных выражений.

Например, конкатенация:

```
'A'.'&SYSLIST(2)'(2, 0).'BC'
```

имеет значение

```
ABC
```

так как подстрока

```
'&SYSLIST(2)'(2, 0)
```

каков бы ни был второй позиционный операнд макрокоманды, всегда имеет пустое значение.

Переменные типа С могут использоваться в левых частях операторов SETC, в символьных выражениях, в арифметических выражениях (при условии, что значение переменной есть изображение десятичного целого без знака). Они могут также использоваться отдельно или в конкатенациях с текстами, параметрами и другими переменными для формирования имени, операции или операндов в модельных предложениях.

6.5.3. Локальные переменные типа В и логические выражения. Локальные переменные макропроцессора типа В могут принимать в качестве значений только двоичные 0 или 1, имеющие смысл логических значений «нет» и «да». Они определяются операторами LCLB (LoCaL Binary), присваивающими переменным начальные значения 0. Обозначения переменных, правила и области определения аналогичны рассмотренным в 6.5.1 и 6.5.2. Новые значения присваиваются переменным типа В операторами SETB, имеющими формат

переменная SETB логическое выражение комментарий

Логическим выражением называется конструкция, определяющая одно из значений: B'0' или B'1'. В простейшем случае оно состоит из символа 0 или 1, который может быть заключен в скобки

```
&B1     SETB    0
&B2     SETB    (1)
```

Более сложные логические выражения строятся из логических термов и обязательно заключаются в скобки. Логический терм — переменная типа В, арифметическое отношение или символьное отношение. Арифметическим (или соответственно символьным) отношением называются два арифметических (соответственно символьных) выражения, соединенные операцией отношения. Для обозначения операции отношения употребляются следующие идентификаторы:

LT	(Less Then)	операция «меньше»
LE	(Less then or Equal)	операция «меньше или равно»
EQ	(EQual)	операция «равно»
NE	(Not Equal)	операция «не равно»
GE	(Greater then or Equal)	операция «больше или равно»
GT	(Greater Then)	операция «больше»

Операция отношения отделяется от соединяемых ею выражений по крайней мере одним пробелом. Значение арифметического отношения определяется по правилам сравнения целых чисел. При вычислении значения символьного отношения символьные значения его компонент сравниваются как тексты. Если компоненты имеют разную длину, короткая считается меньше длинной. Примеры логических выражений из одного терма:

```
(&A1 + &A2 LE 2 * &A3)
('ABC' EQ '&C1'(2, 3))
```

Логические выражения можно строить из нескольких термов с помощью одноместной логической операции NOT или двухместных AND и OR по таким же правилам, как логические выражения языка АЛГОЛ-60. Идентификатор логической операции должен отделяться от терма, если последний не заключен в скобки, и от другой логической операции по крайней мере одним пробелом. Стандартный порядок выполнения логических операций может быть изменен с помощью скобок, количество уровней которых не должно превышать 5. Примеры логических выражений из двух термов:

```
(&A LE 5 OR '&C' EQ '+')
(NOT &B AND NOT &A GT 25)
```

Переменные типа B могут использоваться в левых частях операторов SETB, в логических, символьных и арифметических выражениях. Включать переменные типа B непосредственно в состав модельных предложений нельзя. В случае необходимости двоичное значение переменной нужно предварительно преобразовать в десятичное с помощью оператора

```
&A      SETA    &B      &A — переменная типа A
                  или
&C      SETC    &B      &C — переменная типа C
```

6.5.4. Операторы и метки перехода. Для того чтобы отметить те операторы или модельные предложения входной программы, к обработке которых макропроцессор переходит, нарушая естественный порядок просмотра, применяется конструкция, называемая меткой перехода макропроцессора. Она состоит из точки и произвольного идентификатора длиной не более семи символов и может располагаться на поле имени почти всех модельных предложений и операторов *. Если поле имени нужного модельного предложения или оператора занято, нужно перед ним

* Исключение составляют только команды: COPY, ISTL, ISEQ и операторы: ACTR, MACRO, LCLA, LCLB, LCLC, GBLA, GBLB, GBLC.

поместить пустой оператор макропроцессора ANOP, помеченный меткой перехода:

```
.JUMP    ANOP              пустой оператор макропро-
*                          цессора
&A       SETA    &A + 1    оператор с занятым полем
*                          имени
```

Метка перехода на поле имени модельного предложения не переносится в выходной текст макропроцессора.

Оператор условного перехода имеет формат

```
[метка перехода]    AIF    логическое выражение и
                           метка перехода
```

Логическое выражение на поле операндов должно быть заключено в скобки, и точка, начинающая метку перехода, должна следовать непосредственно за закрывающей скобкой. Результат исполнения оператора AIF — либо переход к предложению, помеченному указанной меткой (когда логическое выражение имеет значение «да»), либо (в случае «нет») продолжение обработки в естественном порядке. Предложение, к которому возможен переход, должно находиться в той же области входного текста, что и оператор AIF (т. е. либо в собственно программе, либо в том же самом макроопределении). Пример оператора AIF

```
          AIF    ('&P' NE '+').MINUS
```

Оператор безусловного перехода осуществляет переход к предложению, помеченному меткой, расположенной на поле операндов. Его формат

```
[метка перехода]    AGO    метка перехода
```

Оператор AGO также не должен выводить за пределы собственно программы или макроопределения.

Рассмотрим несколько примеров применения описанных средств.

Пример 6. Пусть в области определения локальной переменной типа A (т. е. в одном из макроопределений или в собственно программе) расположена следующая последовательность предложений, состоящая из четырех операторов и одного модельного предложения:

```
&A      SETA    0
.L      ANOP
L&A     DC      H'&A'
&A      SETA    &A + 1
        AIF     (&A LE 10).L
```

Тогда в результате ее обработки макропроцессором в выходном тексте появится фрагмент из одиннадцати команд DC:

```
L0      DC      H'0'
L1      DC      H'1'
...
L10     DC      H'10'
```

Пример 7. Предположим, что мы хотим определить макрокоманду с прототипом

```
        IF      &F, &COND, &GOADDR
```

расширение которой обеспечивает условный переход в машинной программе по адресу, замещающему параметр &GOADDR, в зависимости от значения отношения, описываемого двумя первыми операндами следующим образом: первый операнд указывает адрес двоичной переменной, составляющей левую компоненту отношения, а второй состоит из идентификатора операции отношения (такого же, как в логических выражениях макропроцессора) и обозначения десятичного целого со знаком или без знака, состоящего не более чем из восьми символов. Примером макрокоманды может служить:

```
        IF      X, LE — 24, LOOP
```

Макроопределение IF можно написать следующим образом:

```
        MACRO
&L      IF      &F, &COND, &GOADDR
        LCLC    &C1, &C2
&L      L       0, &F           первая модельная ко-
*                               манда
&C1     SETC    '&COND'(1, 2)   выделение идентифи-
*                               катора отношения
&C2     SETC    '&COND'(3, 8)   выделение обозначе-
*                               ния числа
        C       0, =F'&C2'      вторая модельная ко-
*                               манда
.* разветвление обработки в зависимости от операции отношения
        AIF     ('&C2' EQ 'LT').LT
        AIF     ('&C2' EQ 'LE').LE
        AIF     ('&C2' EQ 'EQ').EQ
        AIF     ('&C2' EQ 'GE').GE
        AIF     ('&C2' EQ 'GT').GT
        BNE     &GOADDR
        AGO     .END
```

```
.LT     BL      &GOADDR
        AGO     .END
.LE     BNH     &GOADDR
        AGO     .END
.EQ     BE      &GOADDR
        AGO     .END
.GE     BNL     &GOADDR
        AGO     .END
.GT     BH      &GOADDR
.END    MEND
```

На основании этого определения приведенная выше макрокоманда заменится расширением:

```
        L       0, X            первая модельная ко-
*                               манда
        C       0, =F'-15'      вторая модельная ко-
*                               манда
        BNH     LOOP
```

Заметим, что первый и третий операнды IF могут быть любыми выражениями, допустимыми на поле второго операнда машинных команд формата RX.

Пример 8. Усовершенствуем определение макрокоманды SUM, рассмотренное в примере 2. Мы хотим, чтобы ее расширение при любом допустимом значении третьего операнда состояло из наименьшего количества машинных команд. Ясно, что при большом количестве слагаемых расширение должно содержать цикл. Воспользовавшись для управления им командой BCT, мы можем свести расширение к шести машинным командам. Соответствующими модельными предложениями, вырабатывающими расширение, будут:

```
                L       0, &BEG         первое сла-
*                                       гаемое
                LA      15, (&NUM-1)*4  индекс по-
*                                       следнего
*                                       слагаемого
LOOP&SYSNDX     A       0, &BEG.(15)    добавление
*                                       очередного
*                                       слагаемого
                SH      15, =H'3'
                BCT     15, LOOP&SYSNDX
                ST      0, &S
```

По сравнению со старым вариантом SUM мы заняли два регистра вместо четырех и получили расширение короче на одну

команду. Но зато цикл теперь состоит из трех команд вместо двух в прежнем варианте.

Если значение параметра & NUM лежит в пределах от 1 до 5, можно обойтись меньшим количеством команд. При одном слагаемом достаточно одной команды:

```
        MVC     &S.(4), &BEG
```

Если же количество слагаемых 2, 3, 4 или 5, то расширение из соответственно трех, четырех, пяти или шести команд можно составить по следующей схеме:

```
        L       0, &BEG          1 команда
        A       0, &BEG + 4      }
        ...                      } 1—4 команды
        A       0, &BEG + 16     }
        ST      0, &S            1 команда
```

При пяти слагаемых мы получаем тоже шесть команд, но ясно, что этот вариант следует предпочесть циклическому.

Усовершенствованное макроопределение:

```
        MACRO
&L      SUM     &S, &BEG, &NUM      значение &NUM
*                                   не меньше 1
        LCLA    &A                  добавка к адресу
        AIF     (&NUM GT 1).NOTONE
.* случай одного слагаемого
&L      MVC     &S.(4), &BEG
        AGO     .END
.NOTONE ANOP            общая часть ветвей
*                       .REPEAT и .LOOP
&L      L       0, &BEG первое слагаемое
        AIF     (&NUM GT 5).LOOP
.* вариант без цикла в машинной программе
.REPEAT ANOP            начало цикла макропроцессора
&A      SETA    &A + 4
        AIF     (&A GE 4*&NUM).LE   выход из
*                                   цикла макро-
*                                   процессора
        A       0, &BEG + &A
        AGO     .REPEAT
.* вариант с циклом в машинной программе
.LOOP       LA      15, (&NUM — 1) * 4
```

```
LOOP&SYSNDX     A       0, &BEG.(15)
                SH      15, =H'3'
                BCT     15, LOOP&SYSNDX
.* общий конец ветвей .REPEAT и .LOOP
.LE             ST      0, &S
.END            MEND
```

В этом макроопределении модельная команда:

```
                A       0, &BEG + &A
```

может обрабатываться макропроцессором от нуля до четырех раз.

6.5.5. Дополнительные средства условной генерации. К дополнительным средствам условной генерации относятся операторы MEXIT, MNOTE и ACTR, системная переменная &SYSECT, глобальные переменные макропроцессора и переменные с индексом.

Оператор MEXIT, не имеющий операндов, служит для указания конца формирования макрорасширения. Например, в последнем варианте макроопределения SUM можно заменить оператором MEXIT операторы безусловного перехода:

```
                AGO     .END
```

При этом можно опустить и метку перехода .END на поле имени оператора MEND.

Оператор MNOTE используется внутри макроопределений для включения в состав листинга диагностических сообщений. Поле имени оператора может быть пустым или содержать метку перехода макропроцессора. Поле операндов имеет следующий формат:

$$\left\{\begin{matrix}\text{код серьезности} \\ * \end{matrix}\right\}, \text{'сообщение'}$$

где фигурные скобки означают альтернативу. Код серьезности — десятичное целое, характеризующее тяжесть отмеченной ошибки. Он может формироваться с помощью параметров или переменных макропроцессора типов A или C. Если код серьезности отсутствует (в этом случае можно также не писать запятую), подразумевается значение 1. Звездочка означает, что сообщение не связано с ошибкой и имеет характер комментария. Операнды MNOTE печатаются в листинге в соответствующем месте макрорасширения. При этом ограничивающие сообщение апострофы опускаются, каждый сдвоенный апостроф или амперсанд внутри сообщения изображается одним знаком апострофа или амперсанда, а все обозначения параметров и переменных замещаются их значениями. При наличии кода серьезности сообщение об операторе MNOTE включается в список ошибок в конце листинга объектной программы, а значение кода серьезности влияет на

значение кода возврата, формируемого ассемблером в регистре 15 при завершении работы.

Чаще всего оператор MNOTE применяется для указания автору программы ошибки в кодировке операнда макрокоманды. Например, начало макроопределения SUM, в котором параметр &NUM может замещаться лишь самоопределенным термом, имеющим значение больше нуля, полезно расширить следующим образом:

```
         MACRO
&L       SUM      &S, &BEG, &NUM
         LCLA     &A
         AIF      (T'&NUM NE 'N').MN
         AIF      (&NUM GE 1).OK
.MN      MNOTE    16, 'значение &NUM недопустимо в тре- *
                  тьем операнде SUM'
         MEXIT
.OK      AIF      (&NUM GT 1).NOTONE
         ...
```

Тогда, если во входном тексте встретится, например, макрокоманда:

```
         SUM      F1, F2, X
```

никакого расширения в объектную программу не поступит, а в листинге появится строка

16, значение X недопустимо в третьем операнде SUM

Оператор ACTR имеет формат:

```
пусто    ACTR     арифметическое выражение.
```

Он служит для предотвращения зацикливания макропроцессора, который учитывает количество сделанных им переходов по операторам AIF и AGO с помощью специальных счетчиков. Такие счетчики заводятся для каждой макрокоманды, как внешней, так и внутренней. Отдельный счетчик обслуживает собственно программу. Счетчик макрокоманды получает начальное значение при входе в соответствующее макроопределение, а счетчик собственно программы — при входе в нее. При каждом переходе из соответствующего счетчика вычитается единица. При достижении нуля счетчиком какой-либо макрокоманды макропроцессор прекращает обработку этой макрокоманды (или целого «гнезда» макрокоманд, если исчерпался счетчик внутренней макрокоманды) и переходит к следующему предложению собственно программы. Если же достиг нуля счетчик программы,

обработка входного текста макропроцессором прекращается и начинается трансляция сгенерированного текста.

Действие оператора ACTR состоит в присваивании значения арифметического выражения в качестве начального значения соответствующего счетчика. В собственно программе он должен располагаться после всех описаний глобальных и локальных переменных, но раньше первой команды CSECT, а в макроопределениях — непосредственно после описаний всех глобальных и локальных переменных. При отсутствии оператора ACTR счетчики получают стандартное значение, зависящее от типа ассемблера. Для ассемблера уровня F оно составляет 4096.

Системная переменная &SYSECT при обработке каждой макрокоманды (как внешней, так и внутренней) имеет своим значением ассемблерное имя той программной или фиктивной секции, в которой расположена обрабатываемая макрокоманда. Переменная обычно служит для генерирования команд продолжения текущей секции в тех макроопределениях, которые содержат модельные команды CSECT или DSECT. Примеры использования &SYSECT см. в [5] или в [18].

Кроме локальных переменных, подробно рассмотренных выше, во входном тексте макропроцессора можно пользоваться глобальными переменными типов A, B, C, которые отличаются от локальных соответствующих типов только способами описания и областями действия. Присваивание новых значений глобальным переменным производят операторы SETA, SETB, SETC. Для описания глобальных переменных служат операторы GBLA, GBLB и GBLC, которые могут располагаться в тех же местах входного текста, что и описания локальных переменных. Однако в каждой группе таких описаний все глобальные описания должны предшествовать всем локальным. Если глобальная переменная, обозначенная некоторым идентификатором, описана как в собственно программе, так и в нескольких макроопределениях, то ее областью действия служит совокупность собственно программы и упомянутых макроопределений. Начальное значение такая переменная получает при входе макропроцессора в собственно программу, а изменение ее значения можно осуществить в любой части области действия. Если область действия глобальной переменной не включает собственно программу, она получает начальное значение при первом обращении к какому-либо макроопределению, входящему в ее область действия.

Глобальные переменные употребляются для связи между макроопределениями друг с другом и с операторами собственно программы во время работы макропроцессора.

Пример 9. Исключение из программы макрокоманд отладочной печати. Предположим, что при отладке некоторой программы PRG мы хотим пользоваться специальной макрокомандой TPR, которая позволяет печатать с помощью системного АЦПУ информацию о ходе исполнения программы PRG. После

завершения отладки все экземпляры TPR из программы нужно удалить. Для этой цели составим макроопределение TPR по следующей схеме:

```
        MACRO
        TPR...                          параметры TPR
        GBLB    &B
        AIF     (NOT &B).END
```

модельные команды, обеспечивающие желаемое расширение

```
.END    MEND
```

Собственно программа должна иметь вид:

```
        GBLB    &B
&B      SETC    1
PRG     CSECT
        ...
        TPR     ...
        ...
        TPR     ...
        ...
        END
```

Область действия глобальной переменной & B состоит из собственно программы и макроопределения TPR. При входе макропроцессора в программу & B получает начальное значение «нет», которое сразу же заменяется на «да». При каждом входе в макроопределение & B сохраняет значение «да» и оператор AIF не выводит в конец макроопределения. В результате каждая макрокоманда заменяется непустым расширением. Если же мы исключим из входного текста оператор:

```
&B      SETB    1
```

ситуация изменится. При каждом входе в макроопределение & B будет сохранять значение «нет», оператор AIF будет осуществлять переход на оператор MEND, т. е. расширение TPR окажется пустым.

Тот же самый эффект будет иметь место, если мы исключим из входного текста два оператора:

```
        GBLB    &B
&A      SETB    1
```

Переменные с индексом представляют удобное средство в тех случаях, когда для преобразования входного текста необходимо большое количество переменных макропроцессора какого-либо типа. Эти переменные могут быть как локальными, так и глобальными. Для описания переменных с индексом и придания им начального значения служат уже рассмотренные операторы

GBLA, GBLB, GBLC, LCLA, LCLB и LCLC. Описание переменной с индексом отличается от описания простой переменной тем, что за ее идентификатором следует десятичное целое в скобках, составляющее верхнюю границу изменения индекса. Наибольшее возможное значение верхней границы ассемблера уровня F есть 2500. Нижняя граница всегда равна единице. Например, описание:

```
        LCLA    &A(10), &X
```

определяет одиннадцать локальных переменных типа A: простую переменную &X и десять переменных с индексом, имеющих обозначения:

```
&A(1),    &A(2), ..., &A(10)
```

Переменные с индексом могут использоваться в входном тексте точно в тех же позициях, что и простые переменные соответствующего типа. В качестве индекса можно пользоваться арифметическими выражениями, значения которых не должны выходить за границы изменения индекса. При описании глобальной переменной с индексом, область действия которой состоит из нескольких частей, необходимо во всех описаниях указывать одну и ту же верхнюю границу.

Пример 10. Пусть начало макроопределения содержит следующую последовательность из семи операторов:

```
          ...
          LCLC    &C(6)
          LCLA    &A
.LOOP     ANOP
&A        SETA    &A + 1
&C(&A)    SETC    'ABCDE'(1, &A)
          AIF     (&A LE 4).LOOP
&C(6)     SETC    'PQ'.'&C(5)'(3, 3)
          ...
```

Первый оператор определяет шесть локальных переменных типа C:

```
&C(1), &C(2), &C(3), &C(4), &C(5) и &C(6)
```

и присваивает им всем пустые значения. Операторы третий, четвертый, пятый и шестой образуют цикл макропроцессора, в результате исполнения которого первые пять из названных переменных получают соответственно значения:

```
A  AB  ABC  ABCD  ABCDE
```

В результате исполнения седьмого оператора переменная &C(6) меняет свое значение с пустого на PQCDE.

УКАЗАТЕЛЬ ЛИТЕРАТУРЫ

1. Вычислительная система IBM/360. Принципы работы. М., 1969. 440 с.
2. Единая система ЭВМ/Под ред. А. М. Ларионова. М., 1974. 134 с.
3. Каналы ввода — вывода ЭВМ ЕС-1020/Под ред. А. М. Ларионова М., 1976. 270 с.
4. Джермейн К. Программирование на IBM/360. М., 1971. 870 с.
5. Программирование на языке ассемблера ЕС ЭВМ. З. С. Брич, В. Н. Воющ, Г. С. Дегтярева, Э. В. Ковалевич. М., 1975. 296 с.
6. Радд У. Программирование на языке ассемблера и вычислительные системы IBM 360 и 370. М., 1979. 592 с.
7. Стэблн Д. Логическое программирование в системе /360. М., 1974. 752 с.
8. Единая система электронных вычислительных машин. Операционная система. Ассемблер. Описание языка. Ц51. 804. 001—01 Д16, 1973.
9. Единая система электронных вычислительных машин. Операционная система. Ассемблер. Руководство программиста. Ц51. 804. 001—01 Д17, 1973.
10. Единая система электронных вычислительных машин. Операционная система. Макрокоманды супервизора и управления данными. Руководство программиста. Ц51. 804. 001—01 Д5. 1973.
11. Алгоритмы и организация решений экономических задач. Вып. 9, М., 1978, 180 с.
12. Лавров С. С. Введение в программирование. М., 1973. 362 с.
13. Супервизор и управление данными. М., 1973. 312 с.
14. Лавров С. С., Гончарова Л. И. Автоматическая обработка данных. Хранение информации в памяти ЭВМ. М., 1971. 160 с.
15. Лебедев В. Н., Соколов А. П. Введение в систему программирования ОС ЕС. М., 1978. 144 с.
16. Единая система электронных вычислительных машин. Операционная система. Редактор связей. Руководство программиста. Ц51. 804. 001—01 Д10. 1973.
17. Единая система электронных вычислительных машин. Операционная система. Язык управления заданиями. Описание языка. Ц51. 804. 001—01 Д2. 1973.
18. Хусаинов Б. С. Макросредства в языке ассемблера ЕС ЭВМ. М., 1978. 94 с.

Таблица внутренних и перфокарточных кодов ЕС ЭВМ

Внутренний код	Карточный код	Символ	Внутренний код	Карточный код	Символ
0100 0000		пробел	1100 0101	12, 5	E
0100 1010	12, 2, 8	[	1100 0110	12, 6	F
0100 1011	12, 3, 8	.	1100 0111	12, 7	G
0100 1100	12, 4, 8	<	1100 1000	12, 8	H
0100 1101	12, 5, 8	(	1100 1001	12, 9	I
0100 1110	12, 6, 8	+	1100 1011	12, 0, 3, 8, 9	И
0100 1111	12, 7, 8	!	1100 1100	12, 0, 4, 8, 9	Й
0101 0000	12	&	1100 1110	12, 0, 6, 8, 9	Л
0101 1010	11, 2, 8	]	1101 0000	11, 0	}
0101 1011	11, 3, 8	$	1101 0001	11, 1	J
0101 1100	11, 4, 8	*	1101 0010	11, 2	K
0101 1101	11, 5, 8	)	1101 0011	11, 3	L
0101 1110	11, 6, 8	;	1101 0100	11, 4	M
0101 1111	11, 7, 8	¬	1101 0101	11, 5	N
0110 0000	11	—	1101 0110	11, 6	O
0110 0001	0, 1	/	1101 0111	11, 7	P
0110 1010	12, 11	|	1101 1000	11, 8	Q
0110 1011	0, 3, 8	,	1101 1001	11, 9	R
0110 1100	0, 4, 8	%	1101 1100	12, 11, 4, 8, 9	П
0110 1101	0, 5, 8	_	1101 1101	12, 11, 5, 8, 9	Я
0110 1110	0, 6, 8	>	1110 0000	0, 2, 8	\
0110 1111	0, 7, 8	?	1110 0010	0, 2	S
0111 1010	2, 8	:	1110 0011	0, 3	T
0111 1011	3, 8	#	1110 0100	0, 4	U
0111 1100	4, 8	@	1110 0101	0, 5	V
0111 1101	5, 8	'	1110 0110	0, 6	W
0111 1110	6, 8	=	1110 0111	0, 7	X
0111 1111	7, 8	"	1110 1000	0, 8	Y
1011 0111	12, 11, 0, 7	Ъ	1110 1001	0, 9	Z
1011 1000	12, 11, 0, 6, 9	Ю	1110 1011	11, 0, 3, 8, 9	У
1011 1010	12, 116, 2, 8	Б	1110 1101	11, 0, 4, 8, 9	Ж
1011 1011	12, 11, 0, 3, 8	Ц	1110 1110	11, 0, 6, 8, 9	Ь
1011 1100	12, 11, 0, 4, 8	Д	1110 1111	11, 0, 7, 8, 9	Ы
1011 1110	12, 11, 0, 6, 8	Ф	1111 0000	0	0
1011 1111	12, 11, 0, 7, 8	Г	1111 0001	1	1
1100 0000	12, 0	{	1111 0010	2	2
1100 0001	12, 1	A	1111 0011	3	3
1100 0010	12, 2	B	1111 0100	4	4
1100 0011	12, 3	C	1111 0101	5	5
1100 0100	12, 4	D	1111 0110	6	6

Внутренний код	Карточный код	Символ	Внутренний код	Карточный код	Символ
1111 0111	7	7	1111 1011	12, 11, 0, 3, 8, 9	Ш
1111 1000	8	8	1111 1100	12, 11, 0, 4, 8, 9	Э
1111 1001	9	9	1111 1101	12, 11, 0, 5, 8, 9	Щ
1111 1010	12, 11, 0, 2, 8, 9	З	1111 1110	12, 11, 0, 6, 8, 9	Ч

Примечания. 1. В графе «Карточный код» указаны номера строк карты, имеющие пробивки в колонке, соответствующие символу. Двенадцать строк карты, рассматриваемые в порядке сверху вниз, имеют следующие номера: 12, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

2. Код 0110 0000 соответствует символу минус.

3. Код 0110 1101 соответствует символу подчеркивания, который располагается выше центра строки и должен печататься после перевода строки, содержащей подчеркиваемый символ.

4. Код 0111 1101 определяет апостроф.

ПРИЛОЖЕНИЕ 2

Таблица машинных команд

Графы таблицы с названием «операция» содержат английские и русские наименования машинных команд системы ЕС ЭВМ.

Графа «мнемонический код операции» содержит наименование соответствующей команды на языке ассемблера.

Графа «машинный код операции» содержит шестнадцатиричное изображение машинного кода.

В графе «формат операндов» показана структура поля операндов на языке ассемблера R1, R2, R3, X2 обозначают номера общих регистров и регистров с плавающей точкой, D1 и D2 — смещения, I2 — непосредственный операнд, L1 и L2 — длины операндов, S1 и S2 — переместимые выражения ассемблера.

В графе «тип команды» указан формат команды. Буквой M отмечены псевдокоманды ассемблера, реализуемые машинными командами BC или BCR с некоторой маской.

В графе «возможные прерывания» для каждой команды указаны все прерывания, которые могут произойти при ее исполнении. В колонках A, S и P крестиками указаны прерывания соответственно по превышению максимального адреса в конфигурации, нарушению спецификации операнда, нарушению защиты памяти. В колонке Ov указаны прерывания по переполнению:

F — переполнение с фиксированной точкой,
D — десятичное переполнение,
E — переполнение в порядке.

Обозначения в колонке «прочие» имеют следующий смысл:

Данные — неправильное десятичное упакованное,
A — привилегированная операция,
B — исчезновение порядка,
C — пропадание знаков,
D — невозможность десятичного деления,
E — невозможность деления с плавающей точкой,
F — невозможность деления с фиксированной точкой,
G — повторная команда EX.

В графе «код условия» приняты следующие обозначения:

H — отсутствие переноса из нулевого бита,
I — перенос из нулевого бита,
J — нулевой результат,
K — ненулевой результат,
L — результат меньше нуля,
M — результат больше нуля,
N — код условия не меняется,
O — переполнение,
P — исчезновение порядка,
Q — переполнение порядка,
R — нулевая мантисса,
S — ноль на последнем поле,
T — отрицательное на последнем поле,
U — положительное на последнем поле,
V — разность равна нулю,
W — разность не равна нулю,
X — разность меньше нуля,
Y — разность больше нуля,
Z — первый операнд равен второму,
AA — первый операнд меньше второго,
BB — первый операнд больше второго,
CC — CSW загружено в память,
DD — канал и подканал не заняты,
EE — канал или подканал заняты,
FF — канал работает в монопольном режиме,
GG — монопольная операция прекращена,
HH — канал неисправен,
II — канал хранит прерывание,
JJ — канал доступен,
KK — канал неисправен,
LL — доступно,
MM — ввод/вывод начался,
NN — ненулевой байт отвечает не последнему байту первого операнда,
OO — ненулевой байт отвечает последнему байту первого операнда,
PP — всем байтам первого операнда отвечают нулевые байты,
QQ — соответствует битам 34 и 35 нового PSW,
RR — соответствует битам 2 и 3 в регистре R1,
SS — старший бит указанного байта был равен 0,
TT — старший бит указанного байта был равен 1,
UU — нулевая маска или равны нулю все выделенные биты,
VV — выделенные биты содержат и нули и единицы,
WW — все выделенные биты суть единицы.

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
1	Add	A	SA	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
2	Add	AR	1A	R1, R2			
3	Add Decimal	AP	FA	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
4	Add Halfword	AH	4A	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
5	Add Logical	AL	5E	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
6	Add Logical	ALR	1E	R1, R2			
7	Add Normalized, Long	AD	6A	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
8	Add Normalized, Long	ADR	2A	R1, R2			
9	Add Normalized, Short	AE	7A	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
10	Add Normalized, Short	AER	3A	R1, R2			
11	Add Unnormalized, Long	AW	6E	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
12	Add Unnormalized, Long	AWR	2E	R1, R2			
13	Add Unnormalized, Short	AU	7E	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
14	Add Unnormalized, Short	AUR	3E	R1, R2			
15	And Logical	N	54	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
16	And Logical	NC	D4	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
1	Сложение	RX	x	x	F			Sum = 0	Sum < 0	Sum > 0	Overflow
2	Сложение	RR			F			Sum = 0	Sum < 0	Sum > 0	Overflow
3	Сложение десятичное	SS	x		D	x	Данные	Sum = 0	Sum < 0	Sum > 0	Overflow
4	Сложение полуслов	RX	x	x	F			Sum = 0	Sum < 0	Sum > 0	Overflow
5	Сложение логическое	RX	x	x				Sum = 0 H	Sum ≠ 0 H	Sum = 0 I	Sum ≠ 0 I
6	Сложение логическое	RR						Sum = 0 H	Sum ≠ 0 H	Sum = 0 I	Sum ≠ 0 I
7	Сложение с нормализацией длинное	RX	x	x	E		B, C	R	L	M	
8	Сложение с нормализацией длинное	RR		x	E		B, C	R	L	M	
9	Сложение с нормализацией короткое	RX	x	x	E		B, C	R	L	M	
10	Сложение с нормализацией короткое	RR		x	E		B, C	R	L	M	
11	Сложение без нормализации длинное	RX	x	x	E		C	R	L	M	
12	Сложение без нормализации длинное	RR		x	E		C	R	L	M	
13	Сложение без нормализации короткое	RX	x	x	E		C	R	L	M	
14	Сложение без нормализации короткое	RR		x	E		C	R	L	M	
15	«и» логическое	RX	x	x				J	K		
16	«и» логическое	SS	x			x		J	K		

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
17	And Logical	NR	14	R1, R2			
18	And Logical Immediate	NI	94	D1 (B1), I2		S1, I2	
19	Branch and Link	BAL	45	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
20	Branch and Link	BALR	05	R1, R2			
21	Branch on Condition	BC	47	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
22	Branch on Condition	BCR	07	R1, R2			
23	Branch on Count	BCT	46	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
24	Branch on Count	BCTR	06	R1, R2			
25	Branch on Equal	BE	47 (BC 8)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
26	Branch on High	BH	47 (BC 2)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
27	Branch on Index High	BXH	86	R1, R3, D2 (B2)		R1, R3, S2	
28	Branch on Index Low or Equal	BXLE	87	R1, R3, D2 (B2)		R1, R3, S2	
29	Branch on Low	BL	47 (BC 4)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
30	Branch if Mixed	BM	47 (BC 4)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
31	Branch on Minus	BM	47 (BC 4)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
32	Branch on Not Equal	BNE	47 (BC 7)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
17	«и» логическое	RR						J	K		
18	«и» логическое непосредственное	SI	x			x		J	K		
19	Переход с возвратом	RX						N	N	N	N
20	Переход с возвратом	RR						N	N	N	N
21	Переход по условию	RX						N	N	N	N
22	Переход по условию	RR						N	N	N	N
23	Переход по счету	RX						N	N	N	N
24	Переход по счету	RR						N	N	N	N
25	Переход по равенству	RX, M						N	N	N	N
26	Переход по больше	RX, M						N	N	N	N
27	Переход, если индекс больше	RS						N	N	N	N
28	Переход, если индекс меньше или равен	RS						N	N	N	N
29	Переход по меньше	RX, M						N	N	N	N
30	Переход по смешанному	RX, M						N	N	N	N
31	Переход по минусу	RX, M						N	N	N	N
32	Переход по неравенству	RX, M						N	N	N	N

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
33	Branch on Not High	BNH	47 (BC 13)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
34	Branch on Not Low	BNL	47 (BC 11)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
35	Branch on Not Minus	BNM	47 (BC 11)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
36	Branch on Not Ones	BNO	47 (BC 14)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
37	Branch on Not Plus	BNP	47 (BC 13)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
38	Branch on Not Zeros	BNZ	47 (BC 7)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
39	Branch if Ones	BO	47 (BC 1)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
40	Branch on Overflow	BO	47 (BC 1)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
41	Branch on Plus	BP	47 (BC 2)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
42	Branch if Zeros	BZ	47 (BC 8)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
43	Branch on Zero	BZ	47 (BC 8)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
44	Branch Unconditional	B	47 (BC 15)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
45	Branch Unconditional	BR	07 (BCR 15)	R2			
46	Compare Algebraic	C	59	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
47	Compare Algebraic	CR	19	R1, R2			
48	Compare Decimal	CP	F9	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
33	Переход по не больше	RX, M						N	N	N	N
34	Переход по не меньше	RX, M						N	N	N	N
35	Переход по не минусу	RX, M						N	N	N	N
36	Переход при не единицах	RX, M						N	N	N	N
37	Переход по не плюсу	RX, M						N	N	N	N
38	Переход по не нулю	RX, M						N	N	N	N
39	Переход при единицах	RX, M						N	N	N	N
40	Переход по переполнению	RX, M						N	N	N	N
41	Переход по плюсу	RX, M						N	N	N	N
42	Переход при нулях	RX, M						N	N	N	N
43	Переход по нулю	RX, M						N	N	N	N
44	Безусловный переход	RX, M						N	N	N	N
45	Безусловный переход	RR, M						N	N	N	N
46	Сравнение алгебраическое	RX	x	x				Z	AA	BB	
47	Сравнение алгебраическое	RR						Z	AA	BB	
48	Сравнение десятичное	SS	x								

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
49	Compare Halfword	CH	49	R1, D2 (X2, B2)	R1, D2 (,B2)	R1, S2 (X2)	R1, S2
50	Compare Logical	CL	55	R1, D2 (X2, B2)	R1, D2 (,B2)	R1, S2 (X2)	R1, S2
51	Compare Logical	CLC	D5	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
52	Compare Logical	CLR	15	R1, R2			
53	Compare Logical Immediate	CLI	95	D1 (B1), I2		S1, I2	
54	Compare, Long	CD	69	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
55	Compare, Long	CDR	29	R1, R2			
56	Compare, Short	CE	79	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
57	Compare, Short	CER	39	R1, R2			
58	Convert to Binary	CVB	4F	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
59	Convert to Decimal	CVD	4E	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
60	Divide	D	5D	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
61	Divide	DR	1D	R1, R2			
62	Divide Decimal	DP	FD	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
63	Divide, Long	DD	6D	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
64	Divide, Long	DDR	2D	R1, R2			

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
49	Сравнение полуслов	RX	x	x				Z	AA	BB	
50	Сравнение логическое	RX	x	x				Z	AA	BB	
51	Сравнение логическое	SS	x	x				Z	AA	BB	
52	Сравнение логическое	RR						Z	AA	BB	
53	Сравнение логическое непосредственное	SI	x					Z	AA	BB	
54	Сравнение, длинное	RX	x	x				Z	AA	BB	
55	Сравнение, длинное	RR		x				Z	AA	BB	
56	Сравнение, короткое	RX	x	x				Z	AA	BB	
57	Сравнение, короткое	RR		x				Z	AA	BB	
58	Перевод в двоичную	RX	x	x			Данные, F	N	N	N	N
59	Перевод в десятичную	RX	x	x				N	N	N	N
60	Деление	RX	x	x			F	N	N	N	N
61	Деление	RR		x			F	N	N	N	N
62	Деление десятичное	SS	x	x		x	Данные, D	N	N	N	N
63	Деление, длинное	RX	x	x	E		B, E	N	N	N	N
64	Деление, длинное	RR		x	E		B, E	N	N	N	N

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
65	Divide, Short	DE	7D	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	S1, S2
66	Divide, Short	DER	3D	R1, R2			
67	Edit	ED	DE	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
68	Edit and Mark	EDMK	DF	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
69	Exclusive Or	X	57	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
70	Exclusive Or	XC	D7	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
71	Exclusive Or	XR	17	R1, R2			
72	Exclusive Or Immediate	XI	97	D1 (B1), I2		S1, I2	
73	Execute	EX	44	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
74	Halve, Long	HDR	24	R1, R2			
75	Halve, Short	HER	34	R1, R2			
76	Halt I/O	HIO	9E	D1 (B1)			
77	Insert Character	IC	43	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
78	Insert Storage Key	ISK	09	R1, R2			
79	Load	L	58	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
80	Load	LR	18	R1, R2			

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
65	Деление, короткое	RX	x	x	E		B, E	N	N	N	N
66	Деление, короткое	RR		x	E		B, E	N	N	N	N
67	Редактирование	SS	x			x	Данные	S	T	U	
68	Редактирование с отметкой	SS	x			x	Данные	S	T	U	
69	Исключающее «или»	RX	x	x				J	K		
70	Исключающее «или»	SS	x			x		J	K		
71	Исключающее «или»	RR						J	K		
72	Исключающее «или» непосредственное	SI	x			x		J	K		
73	Выполнить	RX	x	x			G	Может быть изменен			
74	Пополам, длинное	RR		x				N	N	N	N
75	Пополам, короткое	RR		x				N	N	N	N
76	Остановить ввод/вывод	SI					A	DD	CC	GG	KK
77	Загрузка байта	RX	x					N	N	N	N
78	Выдача ключа памяти	RR	x	x			A	N	N	N	N
79	Загрузка	RX	x	x				N	N	N	N
80	Загрузка	RR						N	N	N	N

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
81	Load Address	LA	41	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
82	Load and Test	LTR	12	R1, R2			
83	Load and Test, Long	LTDR	22	R1, R2			
84	Load and Test, Short	LTER	32	R1, R2			
85	Load Complement	LCR	13	R1, R2			
86	Load Complement, Long	LCDR	23	R1, R2			
87	Load Complement, Short	LCER	33	R1, R2			
88	Load Halfword	LH	48	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
89	Load, Long	LD	68	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
90	Load, Long	LDR	28	R1, R2			
91	Load Multiple	LM	98	R1, R3, D2 (B2)		R1, R3, S2	
92	Load Negative	LNR	11	R1, R2			
93	Load Negative, Long	LNDR	21	R1, R2			
94	Load Negative, Short	LNER	31	R1, R2			
95	Load Positive	LPR	10	R1, R2			
96	Load Positive, Long	LPDR	20	R1, R2			

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
81	Загрузка адреса	RX						N	N	N	N
82	Загрузка с проверкой	RR						J	L	M	
83	Загрузка с проверкой, длинная	RR		x				R	L	M	
84	Загрузка с проверкой, короткая	RR		x				R	L	M	
85	Загрузка дополнения	RR			F			P	L	M	O
86	Загрузка дополнения, длинная	RR		x				R	L	M	
87	Загрузка дополнения, короткая	RR		x				R	L	M	
88	Загрузка полуслова	RX	x	x				N	N	N	N
89	Загрузка, длинная	RX	x	x				N	N	N	N
90	Загрузка, длинная	RR	x					N	N	N	N
91	Загрузка многократная	RS	x	x				N	N	N	N
92	Загрузка отрицательного	RR						J	L		
93	Загрузка отрицательного, длинная	RR		x				R	L		
94	Загрузка отрицательного, короткая	RR		x				R	L		
95	Загрузка положительного	RR			F			J		M	O
96	Загрузка положительного, длинная	RR		x				R	L	M	

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
97	Load Positive, Short	LPER	30	R1, R2			
98	Load PSW	LPSW	82	D1 (B1)			
99	Load, Short	LE	78	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
100	Load, Short	LER	38	R1, R2			
101	Move Character	MVC	D2	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
102	Move Immediate	MVI	92	D1 (B1), I2		S1, I2	
103	Move Numerics	MVN	D1	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
104	Move with Offset	MVO	F1	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
105	Move Zones	MVZ	D3	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
106	Multiply	M	5C	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
107	Multiply	MR	1C	R1, R2			
108	Multiply Decimal	MP	FC	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
109	Multiply Halfword	MH	4C	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
110	Multiply, Long	MD	6C	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
111	Multiply, Long	MDR	2C	R1, R2			
112	Multiply, Short	ME	7C	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
97	Загрузка положительного, короткая	RR		x				R	L	M	
98	Загрузка PSW	SI	x	x			A	QQ	QQ	QQ	QQ
99	Загрузка, короткая	RX	x	x				N	N	N	N
100	Загрузка, короткая	RR		x				N	N	N	N
101	Пересылка байтов	SS	x			x		R	N	N	N
102	Пересылка непосредственного	SI	x			x		N	N	N	N
103	Пересылка цифровых полубайтов	SS	x			x		N	N	N	N
104	Пересылка со смещением	SS	x			x		N	N	N	N
105	Пересылка зонных полубайтов	SS	x			x		N	N	N	N
106	Умножение	RX	x	x				N	N	N	N
107	Умножение	RR		x				N	N	N	N
108	Умножение десятичное	SS	x	x		x	Данные				
109	Умножение полуслова	RX	x	x				N	N	N	N
110	Умножение, длинное	RX	x	x	E		B	N	N	N	N
111	Умножение, длинное	RR		x	E		B	N	N	N	N
112	Умножение, короткое	RX	x	x	E		B	N	N	N	N

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
113	Multiply, Short	MER	3C	R1, R2			
114	No Operation	NOP	47 (BC 0)	D2 (X2, B2)	D2 (, B2)	S2 (X2)	S2
115	No Operation	NOPR	07 (BCR 0)	R2			
116	Or Logical	O	56	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
117	Or Logical	OC	D6	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
118	Or Logical	OR	16	R1, R2			
119	Or Logical Immediate	OI	96	D1 (B1), I2		S1, I2	
120	Pack	PACK	F2	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
121	Read Direct	RDD	85	D (B1), I2		S1, I2	
122	Set Program Mask	SPM	04	R1			
123	Set Storage Key	SSK	08	R1, R2			
124	Set System Mask	SSM	80	D1 (B1)		S1	
125	Shift Left Double Algebraic	SLDA	8F	R1, D2 (B2)		R1, S2	
126	Shift Left Double Logical	SLDL	8D	R1, D2 (B2)		R1, S2	
127	Shift Left Single Algebraic	SLA	8B	R1, D2 (B2)		R1, S2	
128	Shift Left Single Logical	SLL	89	R1, D2 (B2)		R1, S2	

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
113	Умножение, короткое	RR		x	E		B	N	N	N	N
114	Пустая операция	RX, M						N	N	N	N
115	Пустая операция	RR, M						N	N	N	N
116	«или» логическое	RX	x	x				J	K		
117	«или» логическое	SS	x			x		J	K		
118	«или» логическое	RR						J	K		
119	«или» логическое непосредственное	SI	x			x		J	K		
120	Упаковка	SS	x			x		N	N	N	N
121	Прямое чтение	SI	x			x		N	N	N	N
122	Замена программной маски	RR					A	RR	RR	RR	RR
123	Замена ключа памяти	RR	x	x			A	N	N	N	N
124	Замена маски системы	SI	x				A	N	N	N	N
125	Сдвиг влево двойной алгебраический	RS		x	F			J	L	M	O
126	Сдвиг влево двойной логический	RS		x				N	N	N	N
127	Сдвиг влево простой алгебраический	RS			F			J	L	M	O
128	Сдвиг влево простой логический	RS						N	N	N	N

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
129	Shift Right Double Algebraic	SRDA	8E	R1, D2 (B2)		R1, S2	
130	Shift Right Double Logical	SRDL	8C	R1, D2 (B2)		R1, S2	
131	Shift Right Single Algebraic	SRA	8A	R1, D2 (B2)		R1, S2	
132	Shift Right Single Logical	SRL	88	R1, D2 (B2)		R1, S2	
133	Start I/O	SIO	9C	D1 (B1)		S1	
134	Store	ST	50	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
135	Store Character	STC	42	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
136	Store Halfword	STH	40	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
137	Store Long	STD	60	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
138	Store Multiple	STM	90	R1, R3, D2 (B2)		R1, R2, S2	
139	Store Short	STE	70	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
140	Subtract	S	5B	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
141	Subtract	SR	1B	R1, R2			
142	Subtract Decimal	SP	FB	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
143	Subtract Halfword	SH	4B	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
144	Subtract Logical	SL	5F	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
129	Сдвиг вправо двойной алгебраический	RS		x				J	L	M	
130	Сдвиг вправо двойной логический	RS		x				N	N	N	N
131	Сдвиг вправо простой алгебраический	RS						J	L	M	
132	Сдвиг вправо простой логический	RS						N	N	N	N
133	Начать ввод/вывод	SI					A	MM	CC	EE	AA
134	Выгрузка	RX	x	x		x		N	N	N	N
135	Выгрузка байта	RX	x			x		N	N	N	N
136	Выгрузка полуслова	RX	x	x		x		N	N	N	N
137	Выгрузка длинного	RX	x	x		x		N	N	N	N
138	Выгрузка многократная	RS	x	x		x		N	N	N	N
139	Выгрузка короткого	RX	x	x		x		N	N	N	N
140	Вычитание	RX	x	x	F			V	X	Y	O
141	Вычитание	RR			F			V	X	Y	O
142	Вычитание десятичное	SS	x		D	x	Данные	V	X	Y	O
143	Вычитание полуслова	RX	x	x	F			V	X	Y	O
144	Вычитание логическое	RX	x	x					W, H	V, I	W, I

Порядковый номер	Операция	Мнемонический код операции	Машинный код операции	Формат операндов			
				явный		неявный	
145	Subtract Logical	SLR	1F	R1, R2			
146	Subtract Normalized, Long	SD	6B	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
147	Subtract Normalized, Long	SDR	6B	R1, R2			
148	Subtract Normalized, Short	SE	7B	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
149	Subtract Normalized, Short	SER	3B	R1, R2			
150	Subtract Unnormalized, Long	SW	6F	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
151	Subtract Unnormalized, Long	SWR	2F	R1, R2			
152	Subtract Unnormalized, Short	SU	7F	R1, D2 (X2, B2)	R1, D2 (, B2)	R1, S2 (X2)	R1, S2
153	Subtract Unnormalized, Short	SUR	3F	R1, R2			
154	Supervisor Call	SVC	0A	I			
155	Test and Set	TS	93	D1 (B1)		S1	
156	Test Channel	TCH	9F	D1 (B1)		S1	
157	Test I/O	TIO	9D	D1 (B1)			
158	Test Under Mask	TM	91	D1 (B1), I2		S1, I2	
159	Translate	TR	DC	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
160	Translate and Test	TRT	DD	D1 (L, B1), D2 (B2)		S1 (L), S2	S1, S2
161	Unpack	UNPK	F3	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2
162	Write Direct	WRD	84	D1 (B1), I2		S1, I2	
163	Zero and Add Decimal	ZAP	F8	D1 (L1, B1), D2 (L2, B2)		S1 (L1), S2 (L2)	S1, S2

Порядковый номер	Операция	Тип команды	Возможные прерывания					Код условия			
			A	S	Ov	P	Прочие	00	01	10	11
145	Вычитание логическое	RR							W, H	V, I	W, I
146	Вычитание с нормализацией, длинное	RX	x	x	E		B, C	R	L	M	
147	Вычитание с нормализацией, длинное	RR		x	E		B, C	R	L	M	
148	Вычитание с нормализацией, короткое	RX	x	x	E		B, C	R	L	M	
149	Вычитание с нормализацией, короткое	RR		x	E		B, C	R	L	M	
150	Вычитание без нормализации, длинное	RX	x	x	E		C	R	L	M	
151	Вычитание без нормализации, длинное	RR		x	E		C	R	L	M	
152	Вычитание без нормализации, короткое	RX	x	x	E		C	R	L	M	
153	Вычитание без нормализации, короткое	RR		x	E		C	R	L	M	
154	Вызов супервизора	RR						N	N	N	N
155	Проверка и запись	SI	x			x		SS	TT		
156	Опрос канала	SI					A	JJ	II	FF	HH
157	Опрос ввода/вывода	SI					A	LL	CC	EE	KK
158	Проверка по маске	SI	x					UU	VV		WW
159	Перекодировка	SS	x			x		N	N	N	N
160	Перекодировка с проверкой	SS	x					PP	NN	OO	
161	Распаковка	SS	x			x		N	N	N	N
162	Прямая запись	SI	x				A	N	N	N	N
163	Очистка и добавление десятичного	SS	x		D	x	Данные	J	L	M	O

Макроопределения EXIT, OPENIN, OPENOUT, OPENSNAP.PROC

```
                                EXIT
         MACRO
&L       EXIT      &REGS=(2,12), &RC=Ø
&L       L         13,4(13)
         MVI       12(13),X'FF'
         L         14,12(13)
         RETURN      &REGS,RC=&RC
         MEND

                               OPENIN
         MACRO
         OPENIN    &NAME,&EXIT
         OPEN      (&NAME)
         CNOP      Ø,4
         B         *+1ØØ
         AIF       ('&EXIT'EQ'').EMP
&NAME    DCB       DSORG=PS,MACPF=(GM),        *
                   DDNAME=&NAME,EODAD=&EXIT
         MEXIT
.EMP     ANOP
&NAME    DCB       DSORG=PS,MACRF=(GM),        *
                   DDNAME=&NAME
         MEND

                              OPENOUT
         MACRO
         OPENOUT   &NAME
         OPEN      (&NAME,(OUTPUT))
         CNOP      Ø,4
         B         *+1ØØ
&NAME    DCB       DSORG=PS,MACRF=(PM),        *
                   DDNAME=&NAME,RECFM=FA,      *
                   LRECL=129,BLKSIZE=129
         MEND

                              OPENSNAP
         MACRO
         OPENSNAP  &NAME
         OPEN      (&NAME,(OUTPUT))
         CNOP      Ø,4
         B         *+92
&NAME    DCB       DSORG=PS,MACRF=(W),         *
                   RECFM=VBA, BLKSIZE=882,     *
                   LRECL=125,DDNAME=&NAME
         MEND
                               PROC

         MACRO
&C       PROC      &R,&RENT=YES,&TM=YES,*
                   &LV=72, &SP=,&ID=
         AIF       ('&R' EQ'').NOCS
&C       CSECT
         USING     &C,&R
         SAVE      (14,12),,&ID
         LR        &R,15
```

```
                AGO     .NOBASE
.NOCS           ANOP
&C              SAVE    (14, 12),, &ID
.NOBASE         ANOP
                AIF     ('&RENT' NE 'YES'). OLD
                AIF     ('&TM' NE 'YES'). GETM
                L       1,8 (13)
                LTR     1,1
                BNZ     B&SYSNDX
.GETM           ANOP
                GETMAIN R, LV=&LV, SP = &SP
                ST      15,8 (1)
                AGO     .NEW
.OLD            ANOP
                CNOP    0,4
                BAL     1, B&SYSNDX
                DC      18F'0'
.NEW            ANOP
B&SYSNDX        ST      13, 4 (1)
                ST      1, 8 (13)
                LM      14, 1, 12 (13)
                L       13, 8 (13)
.OUT            MEND
```

ОГЛАВЛЕНИЕ

ИБ № 1512

Александр Николаевич Балуев

ЭЛЕМЕНТЫ ПРОГРАММИРОВАНИЯ В СИСТЕМЕ ЕС ЭВМ

Редактор *З. И. Царькова*
Художественный редактор *А. Г. Голубев*
Технический редактор *Г. М. Иванова*
Корректоры *С. К. Школьникова, Н. П. Подгородецкая*

Сдано в набор 15.10.81. Подписано в печать 22.10.82. М-13533. Формат 60×90¹/₁₆. Бумага тип. № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 13. Усл. кр.-отт. 13,19. Уч.-изд. л. 12,31. Тираж 27 000 экз. Заказ № 817. Цена 40 коп.

Издательство ЛГУ им. А. А. Жданова, 199164, Ленинград, В-164, Университетская наб., 7/9.

Набрано и сброшюровано в Ленинградской типографии № 2 головного предприятия ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52. Измайловский проспект, 29.

Отпечатано с матриц издательства ЛГУ им. А. А. Жданова, 199164, Ленинград, В-164, Университетская наб. 7/9 в ордена Трудового Красного Знамени Ленинградской тип. № 5 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 190000, Ленинград, центр, Красная ул., 1/3.