В. Н. Пильщиков

Программирование на языке ассемблера IBM PC

Москва • ДИАЛОГ-МИФИ • 2014

УДК 681.325.5
ББК 32.973
П32

Пильщиков В. Н.
П32 Программирование на языке ассемблера IBM PC. – М.: Диалог-МИФИ, 2014. – 288 с.

ISBN 5-86404-051-7

Книга представляет собой учебное пособие по языку ассемблер для персональных компьютеров типа IBM PC. Подробно рассмотрены команды, конструкция языка и методы программирования на нем. Изложение сопровождается примерами.

Для студентов и преподавателей вузов, для всех желающих детально изучить язык ассемблер и приемы программирования на нем.

УДК 681.325.5
ББК 32.973

Учебно-справочное издание
Пильщиков Владимир Николаевич
Программирование на языке ассемблера IBM PC

Редактор О. А. Голубев
Макет Н. В. Дмитриевой

Подписано в печать 1.11.2013
Формат 60x84/16. Бум. офс. Печать офс. Гарнитура Таймс.
Усл. печ. л. 16,74. Уч.-изд. л. 14,64. Тираж 500 экз. Заказ 263
ООО “Издательство Диалог-МИФИ”
115409, Москва, ул. Москворечье, 31, корп. 2. Тел.: 8-905-769-16-61.
Http: //www.dialog-mifi.ru. E-mail: zakaz@dialog-mifi.ru.

Отпечатано ООО "ИНСОФТ"
117105, г. Москва, Варшавское ш., д. 37А

ISBN 5-86404-051-7

ПРЕДИСЛОВИЕ

Данная книга представляет собой учебное пособие по языку ассемблера для персональных компьютеров (ПК), построенных на базе микропроцессоров фирмы Intel, и методам программирования на этом языке. Книга написана на основе лекций, читаемых автором по курсу "Архитектура ЭВМ и язык ассемблера" на факультете вычислительной математики и кибернетики МГУ им. М. В. Ломоносова; учтен в ней и опыт практических занятий по этому курсу.

Хотя язык ассемблера относительно редко используется на практике, его изучение является необходимой частью подготовки профессиональных программистов, поскольку позволяет лучше понять принципы работы ЭВМ, операционных систем и трансляторов с языков высокого уровня, позволяет, в случае необходимости, разработать высокоэффективные программы.

В настоящее время в большинстве ПК используются процессоры фирмы Intel (8086/8088, 80186, 80286, i386, i486, Pentium). Особенностью этих процессоров является преемственность на уровне машинных команд: программы, написанные для младших моделей процессоров, без всяких изменений могут быть выполнены на более старших моделях. При этом базовой является система команд процессора 8086, знание которой является необходимой предпосылкой для изучения остальных процессоров. В книге подробно рассматриваются особенности и система команд именно процессора 8086. При этом предполагается, что читатель уже имеет общее представление об ЭВМ (об их структуре, машинном языке, двоичной системе счисления и т. п.), ибо задача книги – познакомить читателя с конкретной ЭВМ, а не с вычислительными машинами вообще.

Язык ассемблера – это символьная форма записи машинного языка, его использование существенно упрощает написание машинных программ. Для одной и той же ЭВМ могут быть разработаны разные языки ассемблера, в частности, предложено несколько таких языков и для рассматриваемых ПК. В книге рассказывается о языке, разработанном фирмой Microsoft и названном языком макроассемблера (сокращенно – MASM); этот язык наиболее известен и широко используется на практике. Отметим, что существует несколько версий самого языка MASM; в книге рассматривается версия 4.0, как наиболее простая и лежащая в основе последующих версий языка MASM и других языков ассемблера (например, языка Турбо Ассемблер [5]).

Основное внимание в книге уделяется методам программирования на языке MASM. В настоящее время практически все программисты первым изучают язык высокого уровня (Паскаль, Си и т. п.), в котором многие проблемы реализации алгоритмов скрыты от их глаз. Переходя затем к программированию на языке ассемблера, они сталкиваются с этими проблемами и не всегда знают, как их решить. Именно на такие проблемы и обращается особое вни-

мание в книге. Рассказ о приемах программирования на языке ассемблера ведется по следующему принципу: берется какая-то структура данных (массив, списки и т. п.) или структура управления (цикл, процедура, рекурсия и т. п.) из языков высокого уровня и показывается, как эту структуру можно реализовать на языке ассемблера. При этом предполагается, что читатель знаком с одним из языков высокого уровня. Желательно, чтобы это был язык Паскаль, т. к. конструкции именно этого языка берутся за основу. Однако особые тонкости этого языка не рассматриваются, поэтому рассказ будет понятным и тем, кто знаком с иным языком высокого уровня.

Рассказ о системе команд ПК, языке ассемблера и методах программирования на нем сопровождается многочисленными поясняющими примерами.

В большинстве книг по языку ассемблера (см., например, [1-4]) большое внимание уделяется применению его для управления различными устройствами ПК (клавиатурой, дисплеем, дисководами и т. п.), что и понятно, т. к. именно в этих случаях обычно и используется язык ассемблера. Однако в данной книге этим приложениям языка уделяется мало внимания, а в основном рассматриваются общие приемы программирования, поскольку для обучаемого будет проще в дальнейшем осуществить переход от общего к частному, чем от частного к общему.

Автор выражает искреннюю благодарность доценту факультета ВМК МГУ В. Г. Абрамову за большую помощь в подготовке данной книги.

В. Н. Пильщиков

1. ОСОБЕННОСТИ ПЕРСОНАЛЬНОГО КОМПЬЮТЕРА

В книге под термином "персональный компьютер" и сокращением ПК мы будем понимать только персональную ЭВМ, созданную на базе микропроцессоров семейства 80x86 фирмы Intel (8086, 80286, i386, i486, Pentium). Именно к ним относятся наиболее широко распространенные в мире персональные компьютеры фирмы IBM и совместимые с ними.

Первый микропроцессор (процессор, реализованный в виде одной интегральной схемы) появился в 1971 г. Его создала фирма Intel, которая с тех пор остается лидером в области разработки микропроцессоров. Этот процессор, работавший с 4-разрядными данными, представлял собой фактически микрокалькулятор. В 1974 г. фирма создала микропроцессор 8080, работавший с 8-разрядными машинными словами и памятью до 64 килобайт (64 Кб); это уже был настоящий центральный процессор универсальной ЭВМ, хотя и очень простой. В 1976 г. появилась первая персональная ЭВМ (т. е. процессор плюс память и устройства ввода-вывода), разработанная фирмой Apple.

В 1978 г. фирма Intel разработала микропроцессор нового поколения – 16-разрядный процессор 8086 с памятью до 1 Мбайт (1 Мб); по своим возможностям он был на уровне малых ЭВМ того времени. В 1979 г. появился его вариант - микропроцессор 8088, который также работал с 16-разрядными словами, но использовал 8-разрядную шину (в процессоре 8086 была 16-разрядная шина), что позволило воспользоваться имевшимися в то время внешними устройствами (дисководами и т. п.) с 8-разрядными соединениями. На базе этого процессора фирма IBM в 1981 г. создала свой первый ПК под названием IBM PC (personal computer). Появление этого ПК сразу привлекло к себе большое внимание, и именно с этого времени началось широкое распространение ПК в мире. Чуть позже (1983 г.) фирма IBM создала усовершенствованную модель ПК – IBM XT (eXtended Technology).

В 1983 г. фирма Intel разработала микропроцессор 80186, но он практически не использовался, т. к. в том же году появился более совершенный микропроцессор 80286. На его основе IBM в 1984 г. построила свой очередной ПК – IBM AT (advanced technology). В процессоре 80286 предусмотрены аппаратные средства для реализации многозадачного режима работы ЭВМ (одновременного выполнения на ЭВМ нескольких программ). Однако в целом возможности этого процессора оказались недостаточными для реального использования такого режима, поэтому процессор 80286 фактически представляет собой просто более быстрый вариант процессора 8086.

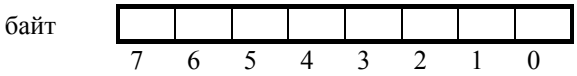
Реально этот режим стал использоваться только с появлением нового поколения микропроцессоров – 32-разрядных. В 1987 г. фирмой Intel был создан процессор i386, а в 1990 г. – процессор i486. Они могут работать в двух режимах – в реальном режиме, в котором они фактически представляют собой очень быстрые варианты процессора 8086, и в защищенном режиме, по-

зволяющем реализовать многозадачность. В 1993 г. фирма Intel разработала 64-разрядный микропроцессор, получивший собственное имя Pentium.

Все указанные процессоры объединяют в семейство 80x86, поскольку в них соблюдается преемственность: программа, написанная для младшей модели, может быть без каких-либо изменений выполнена на любой более старшей модели. Обеспечивается это тем, что в основе всех этих процессоров лежит система команд процессора 8086, в старшие же модели лишь добавляются новые команды (главным образом, необходимые для реализации многозадачного режима). Таким образом, процессор 8086 – это база, основа для изучения всех остальных моделей данного семейства. Именно эта база нас и будет интересовать (многозадачный режим мы рассматривать не будем). Поэтому в дальнейшем под сокращением ПК будем понимать персональный компьютер с процессором 8086.

1.1. Оперативная память

Оперативная память ПК делится на ячейки размером в 8 разрядов. Ячейки такого размера принято называть байтами (byte). Разряды байта нумеруются справа налево от 0 до 7:



При этом правые разряды (с меньшими номерами) называются младшими, а левые разряды – старшими. В каждом разряде может быть записана величина 1 или 0, такую величину принято называть битом (bit). Таким образом, содержимое любого байта – это набор из 8 бит, из 8 нулей и единиц.

Ради краткости договоримся в дальнейшем записывать содержимое ячеек не в двоичной системе, а в шестнадцатеричной, указывая в конце записи числа букву h (hexadecimal – шестнадцатеричный), чтобы отличать такие числа от десятичных. Например, если содержимым байта является 00010011, то будем записывать его как 13h (десятичное 19).

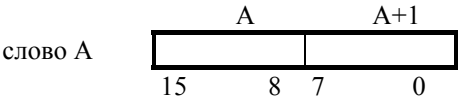
Байты нумеруются начиная с 0, порядковый номер байта называется его адресом. Объем оперативной памяти ПК – 2^{20} байт (1 Мб), поэтому для ссылок на байты памяти нужны 20-разрядные адреса – от 00000h до FFFFFh.

Отметим, что в данной книге на всех рисунках, изображающих память, байты с меньшими адресами будут располагаться вверху (или слева), а с большими адресами – внизу (или справа):

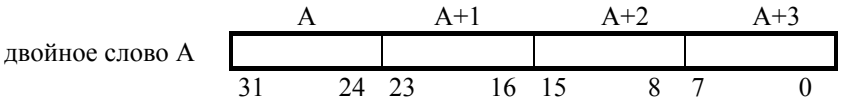
Это следует учитывать, поскольку во многих книгах по ПК память на рисунках изображается в обратном порядке, а нередко и в обоих направлениях, из-за чего часто возникает путаница.

Байт – это наименьшая адресуемая ячейка памяти. Но в ПК имеются и более крупные адресуемые ячейки – слова и двойные слова.

Слово (word) – это два соседних байта. Размер слова – 16 разрядов. Они нумеруются, если рассматривать слово как единое целое, справа налево от 0 до 15. Адресом слова считается по определению адрес его первого байта (с меньшим адресом).



Двойное слово (double word) – это четыре соседних байта или, что то же самое, два соседних слова. Размер двойного слова – 32 разряда, они нумеруются справа налево от 0 до 31. Адрес двойного слова – адрес первого из его байтов (с наименьшим адресом).



ПК может работать как с байтами, так и со словами и двойными словами, т. е. в ПК имеются команды, в которых ячейки этих размеров рассматриваются как единое целое. В то же время слова и двойные слова можно обрабатывать и побайтно.

Отметим, что адрес ячейки еще не однозначно определяет ячейку, поскольку с этого адреса может начинаться ячейка размером в байт, ячейка размером в слово и ячейка размером в двойное слово. Поэтому нужно еще тем или иным способом указывать размер ячейки. Как это делается, мы увидим позже.

Зачем введены ячейки разных размеров? Это связано с тем, что данные разных типов имеют разные размеры, поэтому и нужны ячейки разных размеров. Например, байты используются для хранения небольших целых чисел (типа счетчиков) и символов. В виде же слов представляют обычные целые чисел и адреса. Двойные слова используются для хранения больших чисел.

1.2. Регистры

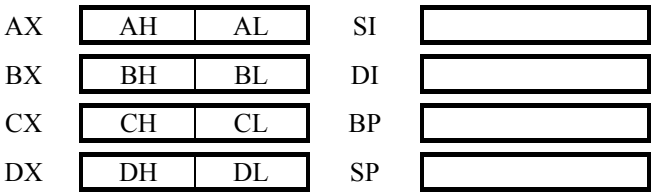
Помимо ячеек оперативной памяти для хранения данных (правда, кратковременного) можно использовать и регистры – ячейки, расположенные в центральном процессоре и доступные из машинных программ. Доступ к регистрам осуществляется намного быстрее, чем к ячейкам памяти, поэтому использование регистров заметно уменьшает время выполнения программ.

Все регистры имеют размер слова (16 разрядов), за каждым из них закреплено определенное имя (AX, SP и т. п.). По назначению и способу использования регистры можно разбить на следующие группы:

- регистры общего назначения (AX, BX, CX, DX, SI, DI, BP, SP);
- сегментные регистры (CS, DS, SS, ES);
- указатель команд (IP);
- регистр флагов (Flags).

1.2.1. Регистры общего назначения

К этой группе относятся следующие 8 регистров:



Хотя названия многих из этих регистров малоосмысленны, все же приведем расшифровку этих названий:

AX accumulator, аккумулятор;
BX base, база;
CX counter, счетчик;
DX data, данные;

(буква X – от слова eXtended, расширенный: в процессоре 8080 были байтовые регистры A, B, C и D, но затем их расширили до размера слова)

SI source index, индекс источника;
DI destination index, индекс приемника;
BP base pointer, указатель базы;
SP stack pointer, указатель стека.

Особенностью всех этих регистров является то, что их можно использовать в любых арифметических, логических и т. п. машинных операциях. Например, можно сложить число из регистра DI с числом из регистра SP или вычесть из содержимого регистра BP содержимое регистра CX.

В то же время каждый из этих регистров имеет определенную специализацию: некоторые команды требуют, чтобы их операнд или операнды обязательно находились в определенных регистрах. Например, команда деления требует, чтобы первый операнд (делимое) находился в регистре AX или в регистрах AX и DX (в зависимости от размера операнда), а команды управления циклом используют регистр CX в качестве счетчика цикла. Такая специализация регистров будет рассматриваться по ходу дела, при описании команд, а пока расскажем о других специализациях этих регистров.

В ПК используется так называемая модификация адресов. Если в команде операнд берется из памяти, тогда сослаться на него можно указав некоторый адрес и некоторый регистр. В этом случае команда будет работать с так называемым исполнительным адресом, который вычисляется как сумма адреса, указанного в команде, и текущего значения указанного регистра. Именно из ячейки с таким адресом команда и будет брать свой операнд. Выгода от такого способа задания операнда заключается в том, что, меняя значение регистра, можно заставить одну и ту же команду работать с разными ячейками памяти, что, например, полезно при обработке массивов, когда одну и ту же команду надо применять к разным элементам массивов. Замена адреса, указанного в команде, на исполнительный адрес называется модификацией адреса, а используемый при этом регистр называется модификатором. Во многих ЭВМ в качестве модификатора можно использовать любой из имеющихся регистров, но вот в ПК модификаторами могут быть только регистры BX, BP, SI и DI. В этом и заключается основная специализация данных регистров. Отметим также, что в ПК модифицировать адрес можно не только по одному регистру, но и по двум сразу. Правда, в этом случае разрешено использовать не любую пару указанных модификаторов, а только такую, где один из регистров – это BX или BP, а другой – это SI или DI. (Более подробно модификация адресов рассматривается в гл. 5.)

Что же касается специализации регистра SP, то он используется при работе со стеком. Стек – это хранилище информации, функционирующее по правилу: первым из стека всегда считывается элемент, записанный в стек последним. Стек полезен во многих случаях, например, для реализации процедур. В ПК имеются команды, поддерживающие работу со стеком. Так вот, в этих командах предполагается, что регистр SP указывает на ячейку стека, в которой находится элемент, записанный в стек последним. (Более подробно работа со стеком рассматривается в гл. 8.)

Теперь отметим еще одну особенность регистров общего назначения. Среди них выделяются регистры AX, BX, CX и DX: они устроены так, что возможен независимый доступ к их старшей и младшей половинам; можно сказать, что каждый из этих регистров состоит из двух байтовых регистров. Обозначают эти половины буквами H (high – выше, старший) и L (low – ниже, младший) и первой буквой из названия регистра: AH и AL – в AX, BH и BL – в BX и т. д. (см. рис. выше). Например, с регистром AX можно работать так: можно записать в него слово (16 бит), затем можно считать только левую половину этого слова (байт из AH), не считывая правую половину, далее можно сделать запись только в AL, не меняя AH. Таким образом, возможен доступ как ко всему регистру AX, так и к любой его половине. Целиком регистр используется при работе с числами, а его половины – при работе с частями чисел или символами.

Отметим, что на части делятся только регистры AX, BX, CX и DX. Запись же в другие регистры и чтение из них осуществляются только целиком.

1.2.2. Сегментные регистры

Вторую группу регистров образуют следующие 4 регистра:

названия которых расшифровываются так:

CS code segment, сегмент команд;
DS data segment, сегмент данных;
SS stack segment, сегмент стека;
ES extra segment, дополнительный сегмент.

Ни в каких арифметических, логических и т. п. операциях эти регистры не могут участвовать. Можно только записывать в них и считывать из них, да и то здесь есть определенные ограничения.

Эти регистры используются для сегментирования адресов, которое является разновидностью модификации адресов и которое используется для сокращения размера команд. Суть дела здесь в следующем.

Если в ЭВМ используется память большого объема, тогда для ссылок на ее ячейки приходится использовать "длинные" адреса, а поскольку эти адреса указываются в командах, то и команды оказываются "длинными". Это плохо, т. к. увеличиваются размеры машинных программ. Сократить размеры команд при "длинных" адресах" можно, например, так. Любой адрес A можно представить в виде суммы B+D, где B – начальный адрес (база) того участка (сегмента) памяти, в котором находится ячейка A, а D – это смещение, адрес ячейки A, отсчитанный от начала этого сегмента (от B). Если сегменты памяти небольшие, тогда и величина D будет небольшой, поэтому большая часть "длинного" адреса A будет сосредоточена в базе B. Этим и можно воспользоваться: если в команде надо указать адрес A, тогда "упрятываем" базу B в какой-нибудь регистр S, а в команде вместо A указываем этот регистр и смещение D. Поскольку для записи D надо меньше места, чем для адреса A, то тем самым уменьшается размер команды. С другой стороны, благодаря модификации адресов данная команды будет работать с адресом, равным сумме D и содержимого регистра S, т. е. с нужным нам адресом A.

Рассмотренный способ задания адресов в командах называется сегментированием адресов (другое название – базирование адресов), а регистры, используемые для хранения начальных адресов сегментов памяти, – сегментными. В ПК в качестве сегментных регистров можно использовать не любой регистр, а только один из следующих четырех: CS, DS, SS и ES.

Более детально сегментирование адресов будет рассмотрено в гл. 7, а пока отметим лишь следующее.

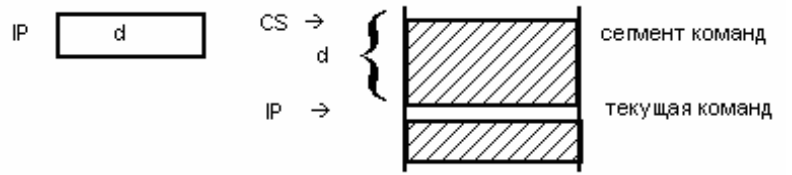
В ПК размеры сегментов памяти не должны превышать 64 Кб (216 = = 65536), поэтому смещения здесь – это 16-разрядные адреса. Поскольку сегментирование адресов применяется в отношении всех команд, операнды ко-

торых берутся из памяти, то в командах явно указываются только 16-разрядные адреса (смещения), а не "длинные" 20-разрядные адреса. Кроме того, в ПК принят ряд соглашений, которые позволяют во многих командах не указывать явно сегментные регистры. В связи с этим во многих программах, особенно небольших, можно ни разу не встретить 20-разрядные адреса и сегментные регистры, и создается впечатление, что ПК – это ЭВМ с 16-разрядными адресами. Учитывая это и не желая вначале усложнять рассказ про ПК сегментированием адресов, договоримся на первых порах считать, что в ПК используются только 16-разрядные адреса, и лишь позже (в гл. 7) мы вспомним, что в ПК настоящие адреса все-таки 20-разрядные. Одновременно договоримся термином "адрес" обозначать 16-разрядные адреса (смещения), указываемые в командах, а 20-разрядные адреса будем называть абсолютными адресами (другое название – физические адреса). Таким образом, адреса меняются от 0000h до FFFFh, а абсолютные адреса – от 00000h до FFFFFh.

Что касается упомянутых соглашений, принятых в ПК, то суть их в следующем: в регистре CS должен находиться начальный адрес сегмента команд – той области памяти, где расположены команды программы; регистр DS должен указывать на начало сегмента данных, в котором размещаются данные программы; регистр SS должен указывать на начало области памяти, отведенной под стек. Если так и сделать, тогда при ссылках на эти сегменты (команд, данных и стека) можно явно не указывать в командах соответствующие сегментные регистры (CS, DS и SS), они будут подразумеваться по умолчанию.

1.2.3. Указатель команд

Еще один регистр ПК – это регистр IP (instruction pointer, указатель команд):



В этом регистре всегда находится адрес команды, которая должна быть выполнена следующей. Более точно, в IP находится адрес этой команды, отсчитанный от начала сегмента команд, на начало которого указывает регистр CS. Поэтому абсолютный адрес этой команды определяется парой регистров CS и IP. Изменение любого из этих регистров есть ничто иное, как переход. Поэтому содержимое регистра IP (как и CS) можно менять только командами перехода.

1.2.4. Регистр флагов

И, наконец, в ПК имеется регистр флагов. Флаг – это бит, принимающий значение 1 ("флаг установлен"), если выполнено некоторое условие, и значение 0 ("флаг сброшен") в противном случае. В ПК используется 9 флагов, причем конструктивно они собраны в один 16-разрядный регистр, называемый регистром флагов и обозначаемый как Flags. Каждый флаг – это один из разрядов данного регистра (некоторые разряды регистра не заняты):

Flags					0F	DF	IF	TF	SF	ZF		AF		PF		CF
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Некоторые флаги принято называть флагами условий; они автоматически меняются при выполнении команд и фиксируют те или иные свойства их результата (например, равен ли он нулю); проверка этих флагов позволяет проанализировать результаты команд. Другие флаги называются флагами состояний; сами по себе они не меняются и менять их должна программа; состояние этих флагов оказывает влияние на дальнейшее поведение процессора.

Флаги условий:

CF (carry flag) – флаг переноса. Наиболее полезен в арифметических операциях над числами без знака; например, если при сложении беззнаковых чисел получилась слишком большая сумма – с единицей переноса, которой нет места в ячейке, тогда флаг CF принимает значение 1, а если сумма "укладывается" в размер ячейки, то значением CF будет 0.

OF (overflow flag) – флаг переполнения. Полезен в арифметических операциях над числами со знаком; например, если при сложении или вычитании знаковых чисел получился результат, по модулю превосходящий допустимую величину (произошло переполнение мантиссы), тогда флаг OF получает значение 1, а если переполнения мантиссы не было – значение 0.

ZF (zero flag) – флаг нуля. Устанавливается в 1, если результат команды оказался нулевым.

SF (sign flag) – флаг знака. Устанавливается в 1, если в операции над знаковыми числами получился отрицательный результат.

PF (parity flag) – флаг четности. Равен 1, если в 8 младших битах результата очередной команды содержится четное количество двоичных единиц. Учитывается обычно только в операциях ввода-вывода.

AF (auxiliary carry flag) – флаг дополнительного переноса. Фиксирует особенности выполнения операций над двоично-десятичными числами.

Флаги состояний:

DF (direction flag) – флаг направления. Устанавливает направление просмотра строк в строковых командах: при DF=0 строки просматриваются "вперед" (от начала к концу), при DF=1 – в обратном направлении.

IF (interrupt flag) – флаг прерываний. При IF=0 процессор перестает реагировать на поступающие к нему прерывания, а при IF=1 блокировка прерываний снимается.

TF (trap flag) – флаг трассировки. При TF=1 после выполнения каждой команды процессор делает прерывание, чем можно воспользоваться при отладке программы – для ее трассировки.

1.3. Представление данных

1.3.1. Представление целых чисел

В общем случае под целое число можно отвести любое число соседних байтов памяти, однако система команд ПК поддерживает работу с числами только размером в байт и слово и частично поддерживает работу с числами размером в двойное слово (если числа занимают иное количество байтов, то все операции над ними надо реализовывать самому программисту). Поэтому можно считать, что в ПК целые числа представляются только байтом, словом или двойным словом. Именно эти форматы чисел мы и будем рассматривать.

В ПК делается различие между целыми числами без знака (неотрицательными) и со знаком. Это объясняется тем, что в ячейках одного и того же размера можно представить больший диапазон беззнаковых чисел, чем неотрицательных знаковых чисел. Например, в байте можно представить беззнаковые числа от 0 до 255, а неотрицательные знаковые числа – только от 0 до 127. Поэтому, если известно заранее, что некоторая числовая величина является неотрицательной, то выгоднее рассматривать ее как беззнаковую, чем как знаковую.

Целые числа без знака

Беззнаковые числа могут быть представлены в виде байта, слова или двойного слова – в зависимости от их размера. Такие числа записываются в ячейки в двоичной системе счисления, занимая все разряды ячейки. Например, если для целого 98 отведен байт, то содержимым байта будет двоичное число 01100010 (62h), а если отведено слово, то оно будет иметь вид 0062h.

Поскольку в ячейке из k разрядов можно записать 2^k различных комбинаций из 0 и 1, то в виде байта можно представить целые от 0 до 255 ($=2^8-1$), в виде слова – целые от 0 до 65535 ($=2^{16}-1$), в виде двойного слова – целые от 0 до 4 294 967 295 ($=2^{32}-1$).

Отметим "экзотическую" особенность представления чисел в ПК: числа размером в слово и двойное слово хранятся в памяти в "перевернутом" виде". Если на число отведено слово памяти, то старшие (левые) 8 бит числа размещаются во втором байте слова, а младшие (правые) 8 бит – в первом байте; в терминах шестнадцатеричной системы: первые две цифры числа хранятся во втором байте слова, а две последние цифры – в первом байте. Например, число 98 = 0062h хранится в памяти так (A – адреса слова):

A	A+1
62	00

Зачем так сделано? Как известно, сложение и вычитание многозначных чисел мы начинаем с действий над младшими цифрами (например, при сложении чисел 1234 и 5678 мы сначала складываем цифры 4 и 8), а затем постепенно переходим к более старшим цифрам. С другой стороны, первые модели ПК (с процессором 8080) были 8-разрядными, в них за раз можно было считать из памяти только один байт. Поскольку в этих условиях многозначное число нельзя считать из памяти сразу целиком, то в первую очередь приходится считывать байт, где находятся младшие цифры числа, а для этого надо, чтобы такой байт хранился в памяти первым. По этой причине в первых моделях ПК и появилось "перевернутое" представление чисел. В последующих же моделях, где уже можно было сразу считать из памяти все число, ради сохранения преемственности, ради того, чтобы ранее составленные программы могли без изменений выполняться на новых ПК, сохранили это "перевернутое" представление чисел.

Отметим, что в регистрах числа размером в слово хранятся в нормальном, неперевернутом виде – за этим следят команды пересылки:

A	A+1
62	00

⇔ BX

BH	BL
00	62

"Перевернутое" представление используется и для чисел размером в двойное слово: в первом байте двойного слова хранятся младшие (правые) 8 бит числа, во втором байте – предпоследние 8 бит и т. д. Например, число 12345678h хранится в памяти так:

A	A+1	A+2	A+3
78	56	34	12

Если рассматривать двойное слово как два слова, тогда можно сказать, что в первом слове хранятся 16 младших (правых) битов числа, а во втором слове – 16 старших (левых) битов, причем каждое из этих слов в свою очередь "перевернуто".

Целые числа со знаком

Эти числа также представляются в виде байта, слова и двойного слова. Как байт можно представить числа от -128 до +127, как слово – от -32768 до +32767, как двойное слово – от -2147483648 до +2147483647.

В ПК знаковые числа записываются в дополнительном коде: неотрицательное число записывается так же, как и беззнаковое число, а отрицательное число x представляется беззнаковым числом $2^k-|x|$, где k – количество разрядов в ячейке, отведенной под число:

$$доп(x)=\begin{cases} x, \text{ если } x >= 0 \\ 2^k-|x|, \text{ если } x < 0 \end{cases}$$

Например, дополнительным кодом числа +98 будет байт 62h или слово 0062h, а дополнительным кодом числа -98 – байт 9Eh (=158=256-98) или слово FF9Eh (=2^{16}-98=10000h-62h).

Приведем еще несколько примеров представления знаковых чисел в дополнительном коде (при ячейке размером в байт):

доп(0) = 0 = 00000000
доп(1) = 1 = 00000001 доп(-1) = 256-1 = 255 = 11111111
доп(2) = 2 = 00000010 доп(-2) = 256-2 = 254 = 11111110
доп(3) = 3 = 00000011 доп(-3) = 256-3 = 252 = 11111101
доп(+126) = 126 = 01111110 доп(-126) = 256-126 = 130 = 10000010
доп(+127) = 127 = 01111111 доп(-127) = 256-127 = 129 = 10000001
доп(-128) = 256-128 = 128 = 10000000

Из этих примеров видно, что в дополнительном коде самый левый бит играет роль знакового: для неотрицательных чисел он равен 0, а для отрицательных – 1.

Как и беззнаковые, знаковые числа размером в слово и двойное слово записываются в памяти в "перевернутом" виде. Например, число -98 как слово будет храниться в памяти таким образом:

A	A+1
9E	FF

При этом знаковый бит оказывается во втором (правом) байте слова.

1.3.2. Двоично-десятичные числа

Обработка числовой информации на ЭВМ, как правило, происходит следующим образом. Поскольку исходные данные и результаты записываются в привычной для людей десятичной системе, а ЭВМ работает с двоичными числами, то при вводе числа переводятся в двоичную систему, после чего происходит обработка двоичных чисел, а при выводе результаты переводятся в десятичную систему. При этом на перевод чисел из одной системы в другую, естественно, тратится время. Но если исходных данных и результатов немного, а их обработка данных занимает значительное время, тогда затраты на переводы не очень заметны.

Однако есть классы задач (например, коммерческие), для которых характерен ввод большого массива числовых данных с последующим применением к ним всего одной-двух арифметических операций и выводом также большого количества результатов. В этих условиях переводы чисел из десятичной системы в двоичную и обратно могут занять львиную долю общих затрат времени, что, конечно, невыгодно. С учетом этого в ПК предусмотрено специальное представление целых чисел, при котором они фактически не отличаются от записи чисел в десятичной системе и которое потому практически не требует перевода чисел из внешнего представления во внутреннее и обратно, и предусмотрены команды арифметических операций над такими

числами. Данное представление чисел называется двоично-десятичным (binary coded decimal, BCD-числа) и строится по следующему принципу: берется десятичная запись числа и каждая его цифра заменяется на четыре двоичные цифры (от 0000 до 1001), обозначающие эту цифру в двоичной системе. Например, число 193 будет представлено так: 0001 1001 0011.

Различие между двоичным и двоично-десятичным представлениями чисел проявляется в том, что если в двоичном представлении за основу берется величина числа (независимо от того, как именно оно вначале было записано), то в двоично-десятичном представлении за основу берется запись числа, причем именно в десятичной системе (если число записать в другой системе, скажем в семиричной, то получилось бы иное представление). При этом однозначные числа (от 0 до 9) записываются одинаково в обоих представления, но уже двузначные числа представляются по-разному: например, число 13 в двоичном виде записывается как 00001011, а в двоично-десятичном – как 00010011.

Достоинством двоично-десятичных чисел, как уже отмечалось, является то, что они практически не требуют перевода из десятичной системы. Дело в том, что ввод чисел осуществляется по цифрам, а двоично-десятичное представление и есть последовательность цифр числа. Недостаток же этого представления заключается в том, что имеющиеся в ПК команды позволяют выполнять арифметические операции только над однозначными двоично-десятичными числами, операции же над многозначными числами приходится уже реализовывать программным способом, что долго. В то же время операции над многозначными двоичными числами реализуются аппаратно, а потому выполняются быстрее.

В дальнейшем мы не будем рассматривать двоично-десятичные числа, и лишь в конце книги (в разд. 14.1) им будет уделено внимание.

1.3.3. О вещественных числах

В ПК нет команд, реализующих арифметические операции над вещественными числами. Это связано с тем, что аппаратная реализация этих операций – вещь дорогая, а при создании ПК во главу угла всегда ставили дешевизну; поэтому, чтобы сократить стоимость ПК, в его систему команд и не включают команды вещественной арифметики.

Как же тогда работать на ПК с вещественными числами? Возможны два решения данной проблемы.

Во-первых, на основе имеющихся команд можно написать процедуры, реализующие арифметические операции над вещественными числами, и в те места программы, где требуется выполнить операции над вещественными числами, нужно вставить обращения к этим процедурам.

Во-вторых, к ПК можно присоединить специальное устройство – арифметический сопроцессор, который умеет выполнять арифметические операции над вещественными числами. Центральный процессор взаимодействует с

этим сопроцессором по следующему сценарию: когда надо выполнить вещественную операцию, центральный процессор посылает сигнал сопроцессору и передает ему соответствующие операнды (в ПК есть специальная команда для этого); сопроцессор выполняет указанную операцию, записывает результат в определенное место и возвращает управление центральному процессору, который после этого продолжает свою работу.

Рассказ про арифметические сопроцессоры – это большая самостоятельная тема, которой мы не будем касаться в данной книге. Поэтому в дальнейшем мы не будем рассматривать вещественные числа и связанные с ними команды ПК и средства языка ассемблера (например, директивы DQ и DT).

1.3.4. Представление символьных данных

Как и любая другая информация, символьные данные должны храниться в памяти ЭВМ в двоичном виде. Для этого каждому символу ставится в соответствие некоторое неотрицательное число, называемое кодом символа, и это число записывается в память ЭВМ в двоичном виде. Конкретное соответствие между символами и их кодами называется системой кодировки.

В ЭВМ, как правило, используются 8-разрядные коды символов. Это позволяет закодировать 256 различных символов, чего вполне достаточно для представления многих символов, используемых на практике. Поэтому для кода символа достаточно выделить в памяти один байт. Так и делают: каждый символ представляется своим кодом, который записывают в один байт памяти.

В ПК обычно используется система кодировки ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информации). Конечно, в ней не предусмотрены коды для букв русского алфавита, поэтому в нашей стране используются варианты этой системы кодировки, в которые включают буквы русского алфавита. Чаще всего, пожалуй, используется вариант, известный под названием "Альтернативная кодировка ГОСТ". Мы не будем приводить эти системы кодировки, а отметим лишь следующие их особенности, которые важно знать при работе с символьными данными.

- Код пробела меньше кода любой буквы и цифры и вообще меньше кода любого графически представимого символа.
- Коды цифр упорядочены по возрастанию и идут без пропусков. Поэтому из неравенств код('0') <= код(c) <= код('9') следует, что c – цифра, и поэтому справедливо равенство код(i)=код('0')+i, где i – число от 0 до 9. Отметим также, что код('0') <> 0.
- Коды больших латинских букв упорядочены согласно алфавиту и также идут без пропусков. Поэтому из неравенств код('A') <= код(c) <= код('Z') следует, что c – большая латинская цифра, и поэтому код i-й по порядку (при нумерации с 0) буквы латинского алфавита равен сумме код('A')+i.
- Все то же самое верно и для малых латинских букв.

- В альтернативной кодировке ГОСТ коды русских букв (больших и малых) упорядочены согласно алфавиту, но если коды больших букв идут без пропусков, то между кодами малых букв 'п' и 'р' вклиниваются коды иных символов.

Что касается машинного представления строк, т. е. последовательностей символов, то под каждую строку отводят нужное число соседних байтов памяти, в которые записывают коды символов, образующих строку. Адрес первого из этих байтов считается адресом строки. Отметим, что эта последовательность кодов записывается в ПК в нормальном, неперевернутом виде. Например, строка 'abc' будет представлена в памяти так (A – адрес строки):

	A	A+1	A+2	
	код(a)	код(b)	код(c)	

1.4. Представление команд

Машинные команды ПК занимают от 1 до 6 байт.

Код операции (КОП) занимает один или два первых байта команды. В ПК столь много различных операций, что для них не хватает 256 различных кодов, которые можно представить в одном байте. Поэтому некоторые операции объединяются в группу и им дается один и тот же КОП, во втором же байте этот код уточняется. Кроме того, во втором байте указываются типы операндов и способы их адресации. В остальных байтах команды указываются ее операнды.

Команды ПК могут иметь от 0 до 2 операндов. Размер операндов – байт или слово (редко – двойное слово). Операнд может быть указан в самой команде (это так называемый непосредственный операнд), либо может находиться в одном из регистров ПК и тогда в команде указывается этот регистр, либо может находиться в ячейке памяти и тогда в команде тем или иным способом указывается адрес этой ячейки. Некоторые команды требуют, чтобы их операнд находился в фиксированном месте (например, в регистре AX), и тогда операнд явно не указывается в команде. Результат выполнения команды помещается в регистр или ячейку памяти, откуда берется один из операндов. Например, большинство команд с двумя операндами реализуют действие

```
op1 := op1 * op2,
```

где op1 – регистр или ячейка памяти, а op2 – непосредственный операнд, регистр или ячейка памяти, а * – операция, заданная КОПом.

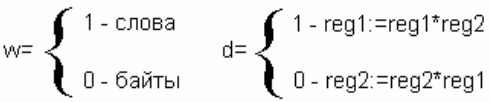
Форматы машинных команд в ПК достаточно разнообразны и "затейливы". Для примера рассмотрим лишь основные форматы команд с двумя операндами.

1) Формат "регистр – регистр" (2 байта):

КОП	d	w
7 2	1	0

11	reg1	reg2
7 6	5 3	2 0

Команды этого формата описывают обычно действие reg1:=reg1*reg2 или reg2:=reg2*reg1, где reg1 и reg2 – регистры общего назначения. Поле КОП первого байта указывает на операцию (*), которую надо выполнить. Бит w определяет размер операндов, а бит d указывает, в какой из двух регистров записывается результат:

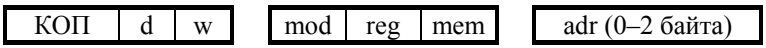


Во втором байте два левых бита фиксированы (для данного формата), а 3-битовые поля reg1 и reg2 указывают на регистры, участвующие в операции, согласно следующей таблице:

reg	*w=1*	*w=0*
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL

reg	*w=1*	*w=0*
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

2) Формат "регистр – память" (2–4 байта):



Эти команды описывают операции reg:=reg*adr или adr:=adr*reg, где reg – регистр, a adr – адрес ячейки памяти. Бит w первого байта определяет размер операндов (см. выше), а бит d указывает, куда записывается результат: в регистр (d=1) или в ячейку памяти (d=0). Трехбитовое поле reg второго байта указывает операнд-регистр (см. выше), двухбитовое поле mod определяет, сколько байтов в команде занимает операнд-адрес (00–0 байтов, 01–1 байт, 10–2 байта), а 3-битовое поле mem указывает способ модификации этого адреса. В следующей таблице указаны правила вычисления исполнительного адреса в зависимости от значений полей mod и mem (a8 – адрес размером в байт, a16 – размером в слово, [r] – содержимое регистра r):

mem \ mod	*00*	*01*	*10*
000	[BX]+[SI]	[BX]+[SI]+a8	[BX]+[SI]+a16
001	[BX]+[DI]	[BX]+[DI]+a8	[BX]+[DI]+a16
010	[BP]+[SI]	[BP]+[SI]+a8	[BP]+[SI]+a16
011	[BP]+[DI]	[BP]+[DI]+a8	[BP]+[DI]+a16
100	[SI]	[SI]+a8	[SI]+a16
101	[DI]	[DI]+a8	[DI]+a16
110	a16	[BP]+a8	[BP]+a16
111	[BX]	[BX]+a8	[BX]+a16

Замечания. Если в команде не задан адрес, то он считается нулевым. Если адрес задан в виде байта (a8), то он автоматически расширяется до слова (a16). Случай mod=00 и mem=110 указывает на отсутствие регистров-

модификаторов, причем адрес должен иметь размер слова. Случай mod=11 соответствует формату "регистр-регистр".

3) Формат "регистр – непосредственный операнд" (3–4 байта):

Команды этого формата описывают операции reg:=reg*im (im – непосредственный операнд). Бит w указывает на размер операндов, а поле reg – на регистр-операнд (см. выше). Поле КОП в первом байте определяет лишь группу операций, в которую входит операция данной команды, уточняет же операцию поле КОП' из второго байта. Непосредственный операнд может занимать 1 или 2 байта (в зависимости от значения бита w), при этом операнд размером в слово записывается в команде в "перевернутом" виде. Ради экономии памяти в ПК предусмотрен случай, когда в операции над словами непосредственный операнд может быть задан байтом (на это указывает 1 в бите s при w=1), и тогда перед выполнением операции байт автоматически расширяется до слова.

4) Формат "память – непосредственный операнд" (3–6 байт):

Команды этого формата описывают операции типа adr:=adr*im. Смысл всех полей – тот же, что и в предыдущих форматах.

Уже из рассмотренных форматов команд видно, что записывать машинные команды ПК в цифровом виде – вещь чрезвычайно неприятная. Сложности возникают и при записи данных; например, знаковые числа приходится представлять в дополнительном коде, а затем еще и "переворачивать". Поэтому нужен какой-то иной, более удобный способ записи команд и данных. И таким способом является язык ассемблера.

2. ЯЗЫК АССЕМБЛЕРА. НАЧАЛЬНЫЕ СВЕДЕНИЯ

Программировать на машинном языке сложно. Одна из причин этого – цифровая форма записи команд и данных, для людей более привычны символьные обозначения. И уже давно придумали средство, упрощающее составление машинных программ. Это язык ассемблера (другое название – автокод), представляющий собой фактически символьную форму записи машинного языка: в нем вместо цифровых кодов операций выписывают привычные знаки операций или их словесные названия, вместо адресов – имена, а константы записывают в десятичной системе. Программу, записанную в таком виде, вводят в ЭВМ и подают на вход специальному транслятору, называемому ассемблером, который переводит ее на машинный язык, и далее полученную машинную программу выполняют.

Как видно, выполнение программы, написанной на языке ассемблера, осуществляется в два этапа: сначала трансляция, затем счет. Это, конечно, увеличивает время, однако на это охотно идут, т. к. составлять программы на языке ассемблера намного проще, чем на машинном языке. Поэтому в настоящее время, если нужно написать машинную программу, ее не пишут на машинном языке, а пишут только на языке ассемблера.

Отметим, что для любой ЭВМ можно придумать разные языки ассемблера, хотя бы потому, что можно по-разному обозначать машинные операции. В частности, и для ПК разработано несколько таких языков (ASM-86, MASM, TASM и др.). Мы будет рассматривать язык, который создан фирмой Microsoft и полное название которого – язык макроассемблера, сокращенно – MASM (смысл приставки "макро" будет понятен позже). Этот язык наиболее часто используется на ПК. Отметим также, что имеется несколько версий самого языка MASM; мы будем рассматривать версию 4.0 как наиболее известную и являющуюся базовой для других языков ассемблера. Именно эту версию мы будем понимать в дальнейшем под словами "язык ассемблера" и сокращением ЯА.

При описании синтаксиса ЯА мы будем использовать формулы Бэкуса-Наура (БНФ) со следующими дополнениями:

- в квадратных скобках будем указывать конструкции, которые можно опускать; например, запись A[B]C означает либо текст ABC, либо текст AC;
- в фигурные скобки будем заключать конструкции, которые могут быть повторены любое число раз, в том числе и ни разу; например, запись A{BC} означает любой из следующих текстов: A, ABC, ABCBC, ABCBCBC и т. д.

2.1. Лексемы

Изучение ЯА начнем с рассказа о том, как в нем записываются лексемы – такие простейшие конструкции, как имена, числа и строки.

2.1.1. Идентификаторы

Идентификаторы нужны для обозначения различных объектов программы – переменных, меток, названий операций и т. п.

В ЯА идентификатор – это последовательность из латинских букв (больших и малых), цифр и следующих знаков:

```
? . @ _ $
```

Причем на эту последовательность накладываются следующие ограничения:

- длина идентификатора может быть любой, но значащими являются только первые 31 символ (идентификаторы, отличающиеся только в 32-й и последующих позициях, считаются одинаковыми);
- идентификатор не должен начинаться с цифры (7A – ошибка);
- точка может быть только первым символом идентификатора (.A – можно, A. – нельзя);
- в идентификаторах одноименные большие и малые буквы считаются эквивалентными (AX, Ax, aX и ax – одно и то же);

Особо подчеркнем, что в идентификаторах нельзя использовать буквы русского алфавита.

Идентификаторы делятся на служебные слова и имена. Служебные слова имеют заранее определенный смысл, они используются для обозначения таких объектов, как регистры (AX, SI и т. п.), названия команд (ADD, NEG и т. п.) и т. п. Все остальные идентификаторы называются именами, программист может пользоваться ими по своему усмотрению, обозначая ими переменные, метки и другие объекты программы.

В качестве имен вообще-то можно использовать и некоторые служебные слова, однако настоятельно не рекомендуется этого делать.

2.1.2. Целые числа

В ЯА имеются средства записи целых и вещественных чисел. Но как уже было сказано в гл. 1, вещественные числа мы не рассматриваем, поэтому здесь рассматриваются только целые числа.

Целые числа могут быть записаны в десятичной, двоичной, восьмиричной и шестнадцатеричной системах счисления (другие системы не допускаются). Десятичные числа записываются как обычно, а вот при записи чисел в других системах в конце числа ставится спецификатор – буква, которая указывает, в какой системе записано это число: в конце двоичного числа ставится буква b (binary), в конце восьмеричного числа – буква o (octal) или буква q (буква "o" очень похожа на ноль, поэтому для меньшей путаницы рекомендуется использовать букву "q"), а в конце шестнадцатеричного числа – буква

h (hexadecimal). Ради общности спецификатор, а именно букву d (decimal), разрешается указывать и в конце десятичного числа, но обычно этого не делают.

Примеры:
Десятичные числа: 25, -386, +4, 25d, -386d.
Двоичные числа: 101b, -11000b.
Восьмиричные числа: 74q, -74q.
Шестнадцатеричные числа: 1AFh, -1AFh.

Сделаем пару замечаний о записи шестнадцатеричных чисел.

Во-первых, если такое число начинается с "буквенной" цифры (A-F), например A5h, тогда становится непонятным, что означает эта запись – число или идентификатор. Чтобы не было путаницы, вводится следующее требование: если шестнадцатеричное число начинается с цифры A-F, то в начале числа обязательно должен быть записан хотя бы один незначащий ноль:

```
0A5h – число,   A5h – идентификатор.
```

Во-вторых, как и в случае идентификаторов, в числах малые и большие буквы отождествляются, поэтому буквы-спецификаторы (h, b и т. д.) и буквенные шестнадцатеричные цифры (A-F) можно записывать как малыми, так и большими буквами. Например, 1Ah, 1ah, 1aH и 1AH – это одно и то же число. В дальнейшем мы будем придерживаться такого правила: буквенные цифры будем записывать большими буквами, а спецификаторы – малыми буквами (например: 1Ah). Так получается наиболее наглядная запись.

2.1.3. Символьные данные

Символы заключаются либо в одинарные, либо в двойные кавычки: 'A' или "A". Естественно, левая и правая кавычки должны быть одинаковыми: 'B" или "B' – ошибка.

Строки (последовательности символов) также заключаются либо в одинарные, либо в двойные кавычки: 'A+b' или "A+B".

Теперь кое-что уточним:

- в качестве символов можно использовать русские буквы;
- в строках одноименные большие и малые буквы не отождествляются ('A+B' и 'a+b' – разные строки);
- если в качестве символа или внутри строки надо указать кавычку, то делается это так: если символ или строка заключена в одинарные кавычки, то одинарную кавычку надо удваивать, а вот двойную кавычку не надо удваивать, и наоборот, если внешние кавычки двойные, то двойная кавычка должна удваиваться, а одинарная не удваивается:

```
'don''t'   'don"t'   "don't"   "don""t"
```

Собственно ради того, чтобы не удваивать внутренние кавычки, в язык и введены два вида кавычек, ограничивающих символы и строки.

2.2. Предложения

Программа на ЯА – это последовательность предложений, каждое из которых записывается в отдельной строке:

```
<предложение>
<предложение>
...
<предложение>
```

Переносить предложение на следующую строку или записывать в одной строке два предложения нельзя. Если в предложении более 131 символа, то 132-й и все последующие символы игнорируются.

При записи предложений действуют следующие правила расстановки пробелов:

- пробел обязателен между рядом стоящими идентификаторами и/или числами (чтобы отделить их друг от друга);
- внутри идентификаторов и чисел пробелы недопустимы;
- в остальных местах пробелы можно ставить или не ставить;
- там, где допустим один пробел, можно ставить любое число пробелов.

Эти правила не относятся к пробелам внутри строк, где пробел – обычный значащий символ.

По смыслу все предложения ЯА делятся на три группы:

- комментарии,
- команды,
- директивы (приказы ассемблеру).

Рассмотрим каждый из этих типов предложений.

2.2.1. Комментарии

Комментарии не влияют на смысл программы, при трансляции ассемблер игнорирует их. Они предназначены для людей, они поясняют смысл программы.

Комментарием считается любая строка, начинающаяся со знака "точка с запятой" (перед ним может быть любое число пробелов) либо пустая строка (точнее, строка, в которой нет иных символов, кроме пробелов). В комментариях можно использовать любые символы, в том числе и русские буквы.

Например, комментариями являются 1-я и 3-я строки в следующем тексте:

```
;это комментарий
ADD AX,0

MOV BX,2
```

Предложения-комментарии обычно используются для пояснения не одной команды (это можно сделать, как увидим, в самой команде), а целой группы команд, следующих за этим комментарием:

```
;вычисление C=НОД(A,B)
...
```

Пустые же строки обычно используются для того, чтобы отделить одну часть программы от другой, чтобы сделать нагляднее деление программы на части.

Отметим, что в ЯА допустим и многострочный комментарий. Он должен начинаться со строчки

```
COMMENT <маркер> <текст>
```

(COMMENT – это одна из директив ЯА). В качестве маркера берется первый за словом COMMENT символ, отличный от пробела; этот символ начинает комментарий. Концом такого комментария считается конец первой из последующих строк программы, в которой (в любой позиции) снова встретился этот же маркер. Например:

```
COMMENT * все
это является
комментарием * и это тоже
```

Такой вид комментария обычно используется, когда надо (например, при отладке) временно исключить из программы некоторый ее фрагмент.

2.2.2. Команды

Предложения-команды – это символьная форма записи машинных команд. Общий синтаксис этого типа предложений таков:

```
[<метка>:] <мнемокод> [<операнды>] [;<комментарий>]
```

Примеры:

```
LAB: ADD SI,2 ;изменение индекса
     NEG A
     CBW
```

Метка

Синтаксически, метка – это имя. Если метка есть, то после нее обязательно ставится двоеточие.

Метка нужна для ссылок на команду из других мест программы, например, для перехода на эту команду. В отличие от машинного языка, где надо высчитывать адреса ячеек, в которые попадают команды, чтобы затем указывать эти адреса в командах перехода, в ЯА достаточно лишь пометить команду и затем ссылаться на нее по метке.

В ЯА разрешается в одной строке указывать только метку (с двоеточием) и больше ничего. Такая метка, считается, метит следующую команду программы. Эта возможность полезна, по крайней мере, в двух случаях: когда команду надо пометить двумя или более метками и когда метка очень длинная и потому остальная часть команды слишком сильно сдвигается вправо, что плохо смотрится.

Пример:

```
INITIALIZATION:
LAB: ADD BX,AX
```

Мнемокод

Мнемокод (мнемонический код) является обязательной частью команды. Это служебное слово, указывающее в символьной форме операцию, которую должна выполнить команда. В ЯА не используются цифровые коды операций, операции указываются только своими символьными названиями, которые, конечно, легче запомнить (слово "мнемонический" означает "легко запоминающийся").

Сами мнемокоды мы будем рассматривать по ходу дела, при описании команд.

Операнды

Операнды команды, если они есть, отделяются друг от друга запятыми. Операнды обычно записываются в виде выражений, о которых мы еще будет говорить. Пока лишь отметим, что частными случаями выражений являются числа и имена переменных. В отличие от машинного языка, в ЯА в командах не указываются адреса ячеек и, чтобы сослаться на какую-то переменную (ячейку), ей дают имя, а затем в командах указывают это имя.

Комментарий

В конце команды можно поместить комментарий, для чего надо поставить точку с запятой и выписать за ней любой текст, который и будет рассматриваться как комментарий. Такой комментарий, в отличие от комментариев-предложений, обычно используется для пояснения именно данной команды.

2.2.3. Директивы

Помимо машинных команд в программе на ЯА надо указывать и другие вещи. Например, надо сообщать, какие константы и переменные используются в программе и какие имена мы им дали. Это делается с помощью предложений, называемых приказами ассемблеру или директивами.

Синтаксис директив следующий:

```
[<имя>] <название директивы> [<операнды>] [;<комментарий>]
```

Пример:

```
X DB 10,-5,0FFh ; массив X
```

Как видно, формат директив в целом совпадает с форматом команд. Единственное отличие – в директиве после имени, если оно есть, не ставится двоеточие.

Имя, указываемое в начале директивы, – это, как правило, имя константы или переменной, описываемой данной директивой. В нашем примере X – это имя переменной-массива из трех элементов.

Названия директив, как и мнемокоды, – служебные слова. При этом заранее известно, какие служебные слова обозначают директивы, а какие – мнемокоды, и путаницы здесь нет. Названия директив будут указываться по мере знакомства с директивами.

Остальные части директивы (операнды и комментарий) записываются так же, как и в командах.

2.2.4. Ссылки назад и вперед

Теперь сделаем несколько замечаний о метках и именах.

Во-первых, метки команд и имена (констант, переменных и т. п.), указываемые в директивах, – это, вообще говоря, разные вещи как по смыслу, так и по ряду формальных признаков. Однако, если не вдаваться в детали, то метки можно рассматривать как имена команд. Поэтому в дальнейшем под термином "имя" мы будем обычно понимать как имена переменных, так и метки.

Во-вторых, появление имени в начале команды или директивы считается описанием данного имени. В ЯА действует общее правило: каждое имя должно быть описано только раз, т. е. в программе не должно быть двух предложений с одним и тем же именем вначале. (Из этого правила есть исключения, они будут оговариваться явно.)

В-третьих, если в языках высокого уровня действует правило "сначала опиши и лишь затем используй", то в ЯА такого правила нет и именем можно пользоваться (ссылаться на него) как до его описания, так и после описания. Поэтому допустимы оба следующих случая:

```
A DB 1 <┐                    NEG A ┌>
  ...   │  ссылка назад        ...  │  ссылка вперед
NEG A  ─┘                  A DB 1 <┘
```

Чтобы различать эти случаи, вводят термины "ссылка назад" и "ссылка вперед". Ссылка назад (см. слева) – это ссылка на имя, которое по тексту программы описано раньше, а ссылка вперед (см. справа) – это ссылка на имя, которое будет описано позже.

При трансляции ассемблер просматривает текст программы на ЯА сверху вниз. Когда он встречает ссылку на имя, которое уже было описано, то он, имея к этому моменту всю информацию об имени, может правильно оттранслировать данную ссылку. Но если ему встретилась ссылка вперед, т. е. имя, которое еще не описано и о котором он пока ничего не знает, то он не всегда может правильно оттранслировать эту ссылку и потому здесь нередко возникают проблемы. В связи с этим, хотя в целом ЯА и допускает ссылки вперед, в некоторых случаях такие ссылки запрещаются (эти случаи будут оговариваться явно), поэтому лучше всего стараться не использовать ссылки вперед.

2.3. Директивы определения данных

Для описания переменных, с которыми работает программа, в ЯА используются директивы определения данных. Одна из них предназначена для описания данных размером в байт, вторая – для описания данных размером в слово, а третья – для описания данных размером в двойное слово. В остальном эти директивы практически не отличаются друг от друга.

2.3.1. Директива DB

По директиве DB (define byte, определить байт) определяются данные размером в байт. Ее синтаксис (без учета возможного комментария в конце) таков:

```
[<имя>] DB <операнд> {,<операнд>}
```

Встречая такую директиву, ассемблер вычисляет операнды и записывает их значения в последовательные байты памяти. Первому из этих байтов дается указанное имя, по которому на этот байт можно ссылаться из других мест программы.

Существует два основных способа задания операндов директивы DB:

- ? (знак неопределенного значения),
- константное выражение со значением от -128 до 255.

Остальные способы задания операндов производны от этих двух.

Операнд «?»

Возможный пример:

```
X DB ?
```

По этой директиве описывается переменная X. Для нее отводится один байт памяти, в который ничего не записывается (в этом байте будет то, что осталось от предыдущей программы, и предугадать, что там находится, нельзя). В этом случае говорят, что переменная не получает начального значения.

Где отводится этот байт? Транслируя программу, ассемблер просматривает предложение за предложением и размещает соответствующие им машинные представления в последовательных ячейках памяти. Поэтому, встречая директиву DB, он отводит под указанную переменную первый из еще не занятых байтов памяти. Это следует учитывать и, например, не надо вставлять директиву DB между командами.

Выделив байт под переменную, ассемблер запоминает его адрес. Когда он снова встретит в тексте программы имя этой переменной, то он заменит его на данный адрес (в этой замене и заключается трансляция имен). Таким образом, в отличие от машинного языка, в ЯА уже не надо следить за адресами ячеек, в которых размещаются переменные, а достаточно дать переменным имена и затем ссылаться на них по этим именам, все остальное сделает за нас ассемблер.

Адрес ячейки, выделенной переменной с именем X, принято называть значением имени X (не путать с содержимым ячейки по этому адресу!). Кроме того, по описанию переменной ассемблер запоминает, сколько байтов занимает переменная в памяти. Этот размер называется типом имени переменной. Значение (адрес) и тип (размер) имени переменной однозначно определяют ячейку, обозначаемую этим именем. Напомним, что с одного и того же адреса в ПК могут начинаться ячейки разных размеров – и байт, и слово, и двойное слово, поэтому кроме начального адреса ячейки надо знать и ее размер. В связи с этим ассемблер запоминает как адрес переменной, так и ее размер.

В языке ЯА имеются так называемые операторы. Это общее название таких понятий, как функции и операции (типа арифметических). Об операторах рассказ впереди, а сейчас рассмотрим лишь один из операторов – оператор типа, который записывается так:

```
TYPE <имя>
```

Значением этого оператора является размер (в байтах) ячейки, выделенной под переменную с указанным именем. Если переменная описана по директиве DB, т. е. как байтовая переменная, то для ее имени значение этого оператора равно 1.

Отметим, что в ЯА есть стандартная константа с именем BYTE и значением 1, поэтому можно записать так:

```
TYPE X = BYTE = 1
```

Операнд – константное выражение со значением от –128 до 255

Мы рассмотрели, как можно описать переменную, которой не присваивается никакого начального значения. Но ЯА позволяет описывать и переменные с начальными значениями. Для этого в качестве операнда директивы DB указывается выражение, которое ассемблер вычислит и значение которого запишет в ячейку, отведенную под переменную. Это и есть начальное значение переменной. Позже, при выполнении программы, его можно будет и изменить, можно будет что-то записать в эту ячейку, но к началу выполнения программы в этой ячейке уже будет находиться данное значение.

В простейшем и наиболее распространенном случае начальное значение байтовой переменной задается в виде числа с величиной от -128 до 255. Например:

```
A DB 254   ; 0FEh
B DB -2    ; 0FEh (=256-2=254)
C DB 17h   ; 17h
```

По каждой из этих директив ассемблер отводит один байт под переменную и записывает в этот байт указанное число. Таким образом, к началу выполнения программы переменная A будет иметь значение 254, переменная B – значение -2, а переменная C – значение 17h.

Операнд-число, естественно, переводится ассемблером в двоичную систему. При этом неотрицательные числа записываются в память как числа без знака, а отрицательные числа записываются в дополнительном коде (см. комментарии к директивам). В связи с этим и получается, что в качестве операндов можно указывать числа от -128 до 255. Отсюда же следует, что числа 254 и -2 будут представлены в памяти одним и тем же байтом 0FEh (это для нас данные числа различны, а для машины они одинаковы, и ей безразлично, что обозначает байт 0FEh – число со знаком или без знака).

В другом распространенном случае в качестве начального значения переменной указывается символ. Такое значение можно задать двояко: либо указать числовой код этого символа, либо указать сам символ в кавычках. Например, в системе кодировки ASCII код символа "*" равен 2Ah, поэтому следующие две директивы эквивалентны:

```
Q DB 2Ah   Q DB "*"
```

Во втором случае ассемблер сам определит код указанного символа и запишет этот код в ячейку памяти. Ясно, что этот вариант лучше – он нагляднее и не требует знания кодов символов, поэтому его обычно и используют на практике.

Мы рассмотрели два основных случая задания начального значения. В общем же случае такое значение указывается любым константным выражением со значением от -128 до 255. (Если значение выходит за эти пределы, то ассемблер зафиксирует ошибку.) Константные выражения аналогичны арифметическим выражениям языков высокого уровня. Мы их рассмотрим позже, а пока лишь отметим, что к таким выражения относится оператор TYPE, поэтому допустима, скажем, следующая директива (имя Q описано выше):

```
V DB TYPE Q
```

которая эквивалентна директиве V DB 1.

Директива с несколькими операндами

Мы рассмотрели случаи, когда в директиве DB указывается один операнд. Это удобно, когда надо описать скалярную переменную, но неудобно, когда надо описать переменную-массив. В самом деле, если надо, к примеру, описать массив из 4 байт с некоторыми начальными значениями, то это можно сделать так:

```
M DB 2
  DB -2
  DB ?
  DB '*'
```

Отметим попутно, что в массивах имя обычно дается только его первому элементу, а остальные оставляют безымянными, поэтому-то в нашем примере имя указано лишь в первой директиве. Если в директиве DB не указано имя, то по ней байт в памяти отводится, но он остается безымянным.

Ясно, что, если в массиве много элементов, то такой способ описания массива слишком громоздок. Поэтому в ЯА допускается упрощенная форма описания массивов, когда он описывается одной директивой, но с несколькими операн-дами – со столькими, сколько элементов в массиве. Например, вместо наших 4 директив можно выписать только одну:

```
M DB 2,-2,?,'*'
```

По директиве DB с несколькими операндами ассемблер выделяет в памяти соседние байты памяти, по одному на каждый операнд, и записывает в эти байты значения операндов (для операнда ? ничего не записывает). В нашем примере ассемблер следующим образом заполнит память:

M

02	FE		2A

Отметим, что имя, указанное в начале директивы, именует только первый из этих байтов. В связи с этим тип имени M равен 1: TYPE M = BYTE. Остальные же байты остаются безымянными. Для ссылок на них в ЯА используются выражения вида M+k, где k – целое число: M+1 – для ссылки на байт со значением FE, M+2 – для ссылки на следующий байт и т. д. Особо подчеркнем, что запись M+1 не следует понимать как сложение содержимого ячейки с именем M (т. е. числа 2) с числом 1. В ЯА запись вида <имя>±k означает, что к адресу указанного имени надо прибавить (или отнять) число k, в результате чего получится некоторый новый адрес, и вот уже по этому адресу и осуществляется доступ к памяти. Таким образом, данная запись означает сложение/вычитание адресов.

Операнд – строка

Возможно еще одно сокращение в директиве DB: если в ней несколько соседних операндов – символы, то их можно объединить в одну строку. Например, следующие две директивы эквивалентны:

```
S DB 'a','b','c'      S DB 'abc'
```

Отметим, что и в этом случае тип имени равен 1 (TYPE S = BYTE), т. к. любая из этих директив является сокращением следующих трех директив:

```
S DB 'a'
  DB 'b'
  DB 'c'
```

а здесь ясно видно, что имя S обозначает только первый байт.

Вопрос о том, объединять соседние символы в одну строку или нет, а если объединять, то какие именно, решает сам автор программы. Например, нашу директиву можно записать и так:

```
S DB 'ab','c'   или   S DB 'a','bc'
```

Операнд – конструкция повторения DUP

Рассмотрим еще одно возможное сокращение в записи директивы DB. Довольно часто в директиве приходится указывать одинаковые операнды. Например, если мы хотим описать байтовый массив R из 8 элементов с начальным значением 0 для каждого из них, то это можно сделать так:

```
R DB 0,0,0,0,0,0,0,0
```

Так вот, эту директиву можно записать и короче:

```
R DB 8 DUP(0)
```

Здесь в качестве операнда использована так называемая конструкция повторения, в которой сначала указывается коэффициент повторения, затем – служебное слово DUP (duplicate, копировать), а за ним в круглых скобках – повторяемая величина.

В общем случае эта конструкция имеет следующий вид:

k DUP (p_1, p_2, ..., p_n)

где k – константное выражение с положительным значением, n>=1, p_i – любой допустимый операнд директивы DB (в частности, это может быть снова конструкция повторения). Данная запись является сокращением для k раз повторенной последовательности указанных в скобках операндов:

$p_1,p_2,\ldots,p_n$, $p_1,p_2,\ldots,p_n$, ... , $p_1,p_2,\ldots,p_n$

k раз

Например, директивы слева эквивалентны директивам справа:

```
X DB 2 DUP('ab',?,1)            X DB 'ab,?,1,'ab',?,1
Y DB -7, 3 DUP(0, 2 DUP(?))     Y DB -7,0,?,?,0,?,?,0,?,?
```

(Тип имен X и Y – BYTE.)

Отметим, что вложенность конструкций DUP можно использоваться для наглядного описания многомерных массивов. Например, директиву

```
A DB 20 DUP(30 DUP(?))
```

можно рассматривать как описание байтовой матрицы A размера 20x30, в которой элементы расположены в памяти следующим образом: первые 30 байт – это элементы первой строки матрицы, следующие 30 байт – это элементы второй строки и т. д.

2.3.2. Директива DW

Директивой DW (define word, определить слово) описываются переменные размером в слово. Она аналогична директиве DB, поэтому лишь вкратце рассмотрим допустимые виды ее операндов.

Операнд ?

Возможный пример:

```
A DW ?
```

По этой директиве ассемблер отводит под переменную A слово памяти, в которое ничего не записывает, т. е. эта переменная не получает начального значения. Тип переменной равен 2, т. к. она занимает два байта. В ЯА есть стандартная константа с именем WORD и значением 2, поэтому данный факт можно записать так:

```
TYPE A = WORD = 2.
```

Константное выражение со значением от –32768 до 65535

Возможные примеры:

```
B DW 1234h
C DW -2
```

По этим директивам под переменные B и C отводится по слову памяти и в эти ячейки записываются указанные числа, которые становятся начальными значениями этих переменных.

Как и в случае директивы DB, неотрицательные числа записываются в память как числа без знака, а отрицательные числа – в дополнительном коде. Поэтому числа, которые могут быть заданы как операнды директивы DW, должны принадлежать отрезку $[-2^{15}, 2^{16}-1]$.

Но здесь имеется и отличие от директивы DB. Напомним, что в ПК числа размером в слово хранятся в памяти в "перевернутом" виде. Так вот, на ЯА такие числа записываются в нормальном, неперевернутом виде, а "переворачиванием" их занимается сам ассемблер, поэтому по нашим двум директивам память заполнится следующим образом:

34	12	FE	FF
B		C	

С учетом этого при программировании на ЯА можно в общем-то забыть о "перевернутом" представлении чисел в памяти ПК.

Частным случаем рассматриваемого вида операнда директивы DW может быть строка из одного или двух символов, например:

```
S1 DW '01'
S2 DW '1'
```

Если указана строка из двух символов, тогда ассемблер берет коды указанных символов (в нашем случае – 30h (код '0') и 31h (код '1')) и образует из них число-слово (3031h), которое и считается начальным значением описываемой переменной (S1). Но как и любое число размером в слово, данное значение будет записано в память в "перевернутом" виде. Если же в правой части директивы DW указан один символ, тогда к нему слева приписывается символ с кодом 0 и дальнейшие действия ассемблера будут такими же, как и в случае двухсимвольной строки. Поэтому по нашим двум директивам память будет заполнена следующим образом:

31	30	31	00
S1		S2	

В связи с тем, что операнды-строки записываются в память в "перевернутом" виде, что в общем-то не характерно для строк, то подобные операнды редко указываются в директиве DW.

Адресное выражение

В качестве операнда директивы DW может быть указано адресное выражение, т. е. выражение, значением которого является адрес. Как записываются такие выражения, мы еще рассмотрим, а пока лишь отметим, что основной случай адресного выражения – это имя переменной или метка. Поэтому допустим такой пример:

```
C DB ?
D DW C
```

В этом случае ассемблер записывает в слово, выделенное под переменную D, адрес переменной C, который становится начальным значением переменной D.

Несколько операндов, конструкция повторения

В правой части директивы DW можно указать любое число операндов, а также конструкцию повторения. Возможный пример:

```
E DW 40000, 3 DUP(?)
```

2.3.3. Директива DD

По директиве DD (define double word, определить двойное слово) описываются переменные, под которые отводятся двойные слова. Поэтому имена этих переменных имеют тип 4 или DWORD (значением этой стандартной константы как раз является число 4). В остальном эта директива похожа на две предыдущие.

Допустимые типы операндов этой директивы таковы.

Операнд «?»

Пример:

```
A DD ?
```

Под переменную A выделяется двойное слово, в которое ассемблер ничего не записывает, т. е. переменная A не получает начального значения.

Целое число со значение от -2^{31} до $2^{32}-1$

Пример:

```
B DD 123456h
```

В данном случае переменная B получает начальное значение, причем это значение ассемблер записывает в память в "перевернутом" виде:

56	34	12	00

B

Константное выражение (со значением от -2^{15} до $2^{16}-1$)

Обратите внимание на диапазон возможных значений выражения – он в два раза меньше диапазона чисел, которые можно записать в двойном слове. Дело в том, что в ЯА все выражения вычисляются в области 16-битовых чисел, т. е. результаты всех операций берутся по модулю 2^{16} (10000h). Поэтому построить выражение, значением которого являлось бы 32-битовое или даже 17-битовое число, не удастся. Единственное исключение – это явно задать в директиве DD "большое" число. Если же мы укажем хотя бы одну операцию, то ответ тут же будет взят по модулю 2^{16}. Например, по директиве

```
X DD 8000h+8002h
```

начальным значением переменной X станет число 2, а не число 10002h.

Конечно, такая особенность задания начальных значений для переменных размером в двойное слово не очень-то приятна, но так уж устроен ЯА, и это надо учитывать.

Адресное выражение

Такой операнд задает абсолютный адрес. Как это делается, будет рассмотрено в гл. 7.

Несколько операндов, конструкция повторения

Возможный пример:

```
DW 33 DUP(?), 12345h
```

2.4. Директивы эквивалентности и присваивания

Мы рассмотрели, как в ЯА описываются переменные. Теперь рассмотрим, как в этом языке описываются константы. Это делается с помощью директивы эквивалентности – директивы EQU (equal, равно), имеющей следующий синтаксис:

```
<имя> EQU <операнд>
```

Здесь обязательно должно быть указано имя, должен быть и операнд, причем только один.

Эта директива аналогична описанию константы в языке Паскаль:

```
const <имя> = <операнд>;
```

Директивой EQU автор программы заявляет, что указанному операнду он дает указанное имя, и требует, чтобы все вхождения этого имени в текст программы ассемблер заменял на этот операнд. Например, если есть директива

```
STAR EQU '*'
```

то ассемблер будет рассматривать предложение

```
T DB STAR
```

как предложение

```
T DB '*'
```

Другими словами, указать имя STAR и указать '*' – это одно и то же.

Отметим, что директива EQU носит чисто информационный характер, по ней ассемблер ничего не записывает в машинную программу. Поэтому директиву EQU можно ставить в любое место программы – и между командами, и между описаниями переменных, и в других местах.

Теперь рассмотрим, каким может быть операнд директивы EQU и в каких случаях полезна эта директива.

Операнд – имя

Если в правой части директивы указано имя регистра, переменной, константы и т. п., тогда имя слева объявляется синонимом данного имени и все последующие вхождения в текст программы этого имени-синонима ассемблер будет заменять на имя, указанное справа. Например:

```
A DW ?
B EQU A
C DW B     ; эквивалентно: C DW A
```

Имена-синонимы обычно используются для введения более удобных, наглядных обозначений. Например, само по себе имя регистра AX ни о чем не говорит, но если мы используем этот регистр для вычисления какой-то суммы, то его можно обозначить SUM:

```
SUM EQU AX
```

и далее использовать это более наглядное имя SUM.

Отметим, что имя, указанное в правой части директивы EQU, может быть описано в программе как до этой директивы, так и после нее.

Операнд – константное выражение

Примеры:

```
 N EQU 100
 K EQU 2*N-1
STAR EQU '*'
```

Если в правой части директивы EQU стоит константное выражение, тогда указанное слева имя принято называть именем константы. Значением этой константы объявляется значение выражения. Например, N – это константа со значением 100, K – со значением 199, а STAR – со значением 2Ah (это код звездочки в системе ASCII). Все последующие вхождения в текст программы имени константы ассемблер будет заменять на значение этой константы. Например, директива

```
X DB N DUP(?)
```

эквивалентна директиве

```
X DB 100 DUP(?)
```

Случаи, когда полезно применение констант, – такие же, как и в языках высокого уровня. Например, в качестве констант рекомендуется описывать размеры массивов, поскольку в таком случае легко настроить программу на

работу с массивом другого размера – для этого достаточно внести изменение лишь в директиву EQU, описывающую константу.

Отметим, что если в константном выражении используются имена других констант, то они должны быть описаны раньше данной директивы EQU, иначе ассемблер, просматривающий текст программы сверху вниз, не сможет вычислить значение этого выражения.

Операнд – любой другой текст

Примеры:

```
 S EQU 'Вы ошиблись'
LX EQU X+(N-1)
WP EQU WORD PTR
```

В данном случае считается, что указанное имя обозначает операнд в том виде, как он записан (операнд не вычисляется). Именно на этот текст и будет заменяться каждое вхождение данного имени в программу. Например, следующие предложения слева эквивалентны предложениям справа

```
ANS DB S, '!'          ANS DB 'Вы ошиблись','!'
    NEG LX                 NEG X+(N-1)
    INC WP [BX]            INC WORD PTR [BX]
```

Такой вариант директивы EQU обычно используется для того, чтобы ввести более короткие обозначения для часто встречающихся длинных текстов. Введя короткое имя, мы далее в программе можем им пользоваться, а уж ассемблер сам будет его заменять на соответствующий текст.

Отметим, что текст, указанный в правой части директивы EQU, должен быть сбалансирован по скобкам и кавычкам и не должен содержать вне скобок и кавычек символа ";". Кроме того, поскольку текст не вычисляется, то в нем можно использовать как имена, описанные до этой директивы EQU, так и имена, описанные после нее.

Теперь рассмотрим еще одну директиву ЯА, похожую на директиву EQU, – директиву присваивания:

```
<имя> = <константное выражение>
```

Эта директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличие от констант, определенных по директиве EQU, данная константа может менять свое значение, обозначая в разных частях текста программы разные числа. Например:

```
K=10
A DW K     ; эквивалентно: A DW 10
K=K+4
B DB K     ; эквивалентно: B DB 14
```

Подобного рода константы можно использовать, например, ради "экономии имен": если в разных частях текста программы используются разные

константы и области использования этих констант не пересекаются, тогда, чтобы не придумывать новые имена, этим константам можно дать одно и то же имя (другими словами, поменять значение константы с таким именем). Однако главное применение таких констант – в макросредствах (см. гл. 11).

Теперь кое-что уточним.

Если с помощью директивы EQU можно определить имя, обозначающее не только число, но и другие конструкции, то по директиве присваивания можно определить только числовую константу. Кроме того, если имя указано в левой части директивы EQU, то оно не может появляться в левой части других директив (его нельзя переопределять). А вот имя, появившееся в левой части директивы присваивания, может снова появиться в начале другой такой директивы (но только такой!). Поэтому ошибочными являются все следующие три фрагмента программы:

```
K EQU 1   K EQU 1   K=1
K EQU 2   K=2       K EQU 2
```

Появление в языке констант, которые могут менять свои значения, вносит некоторую неопределенность. Рассмотрим, к примеру, фрагмент слева:

```
K=1                 K=1
N EQU K             N EQU K+10
A DW N   ;A=1       C DW N     ;C=11
K=2                 K=2
B DW N   ;B=2       D DW N     ;D=11
```

Какие начальные значения получат переменные A и B? Относительно значения переменной A сомнений нет – это 1. Но вот значением переменной B, оказывается, будет число 2, а не 1. Почему? Дело в том, что имя N объявлено синонимом имени K, поэтому все вхождения имени N ассемблер будет заменять на имя K. Значит, директива B DW N понимается как директива B DW K. Но в этом месте текста программы константа K имеет значение 2, поэтому данная директива воспринимается как B DW 2.

В то же время в примере справа переменные C и D получат одно и то же значение 11. Дело в том, что в директиве EQU указано, так сказать, настоящее константное выражение, и потому ассемблер вычисляет его сразу. Значение 11 этого выражения объявляется значением константы N, которая далее и будет обозначать это число.

Таким образом, необходимо внести следующее уточнение в действие директивы EQU: если в правой части директивы указано имя константы, то имя слева надо понимать не как имя константы (не как имя числа), а как синоним имени справа; если же в правой части указано любое другое константное выражение, тогда имя слева действительно становится именем константы (обозначением числа). Что же касается директивы присваивания, то ее правая часть всегда вычисляется сразу и полученное число тут же становится новым значением константы.

2.5. Выражения

Операнды директив, как правило, описываются в виде выражений. Выражения используются и для описания операндов команд.

В целом выражения ЯА похожи на арифметические выражения языка высокого уровня, однако между ними есть и отличия. Наиболее важное отличие заключается в том, что выражения ЯА вычисляются не во время выполнения программы, а во время ее трансляции: встретив в тексте программы выражение, ассемблер вычисляет его и полученное значение (например, число) записывает в машинную программу. Поэтому, когда программа начнет выполняться, от выражений не останется никаких следов. В связи с этим в выражениях ЯА можно использовать только такие величины, которые известны на этапе трансляции (например, адреса и типы имен), и ни в кое случае нельзя использовать величины (например, содержимое регистров или ячеек памяти), которые станут известными лишь во время счета программы.

Вообще говоря, в ЯА для записи операндов директив и команд достаточно только чисел и имен. Более сложные выражения являются лишь удобной формой записи этих чисел и имен. Например, если имеется переменная X размером в слово и нужно описать переменную TX, начальным значением которой является тип (размер) этой переменной, то вместо директивы

```
TX DB 2
```

лучше использовать эквивалентную ей директиву

```
TX DB TYPE X
```

поскольку она более универсальна и ее не надо менять, если изменится тип переменной X.

В ЯА выражения делятся на два класса – на константные и адресные, в зависимости от типа их значений. Если значением выражения является целое число, то оно называется константным выражением, а если значением является адрес, то это адресное выражение. Конечно, адрес – это тоже целое число (порядковый номер ячейки в памяти), но по смыслу адреса отличаются от просто чисел, и в ЯА они рассматриваются как самостоятельный тип данных.

Отметим, что выражений иных типов в ЯА нет, однако имеются объекты (например, строки и имена регистров), которые не относятся ни к константным, ни к адресным выражениям, но которые также используются для записи операндов команд и директив.

По структуре и назначению выражения, константные и адресные, можно разделить на простейшие выражения и операторы. Из простейших выражений строятся любые другие выражения. К простейшим выражениям относятся числа, имена констант, имена переменных и т. п. Термином "оператор" в ЯА принято обозначать то, что мы обычно называем функциями и операциями. Операторы ЯА делятся на одноместные (это аналог функций одного аргумента) и двухместные (это аналог бинарных операций); примером может служить TYPE X или A+1. С помощью операторов из простейших выражений строятся более сложные выражения.

С простейшими выражениями и операторами мы будем знакомиться по ходу дела. Однако уже сейчас укажем старшинство всех операторов ЯА (в порядке убывания; в каждой строке указаны операторы одного старшинства):

1. (), [], LENGTH, SIZE, WIDTH, MASK
2. .
3. :
4. PTR, OFFSET, SEG, TYPE, THIS
5. HIGH, LOW
6. одноместные + и -
7. *, /, MOD, SHL, SHR
8. двухместные + и -
9. EQ, NE, LT, LE, GT, GE
10. NOT
11. AND
12. OR, XOR
13. SHORT, .TYPE

Операторы одного старшинства вычисляются слева направо. Например, A+B-C – это (A+B)-C.

2.5.1. Константные выражения

Значениями константных выражений всегда являются 16-битовые целые числа (единственным исключением является директива DD, в которой можно явно указывать 32-битовые числа). При этом, если значением (в математическом смысле) константного выражения является отрицательное число, то в формируемую машинную программу оно записывается в дополнительном коде.

К простейшим константным выражениям относятся:

- число от -2^{15} до $2^{16}-1$ (числа вне этого диапазона рассматриваются как ошибка);
- символ (значением такого выражения является код символа; например, значением выражения '0' будет число 30h);
- строка из двух символов (значением является число-слово, составленное из кодов этих символов; например, значением выражения '01' будет число 3031h);
- имя константы (значением такого выражения является значение константы; например, если была директива K EQU 10, то значение K равно 10).

Другие простейшие константные выражения будут рассмотрены позже.

Среди константных операторов, т. е. операторов с числовым значением, пока отметим уже известный нам оператор TYPE и следующие арифметические операторы (k, k1 и k2 означают любые константные выражения):

- одноместные плюс и минус: +k, -k
- операторы сложения и вычитания: k1+k2, k1-k2
- операторы умножения и деления: k1*k2, k1/k2, k1 MOD k2
  (* – умножение, / – деление нацело, MOD – взятие остатка от деления)

Пример использования константного выражения с арифметическими операторами:

```
K EQU 30
X DB (3*K-1)/2 DUP(?) ;массив из 44 байт
```

Операндами арифметических операторов должны быть константные выражения. Но здесь есть одно важное исключение: в ЯА разрешается вычитание одного адреса из другого и в результате получается число, а не адрес.

Пример:

```
    X DW 1,2,3,4,5
    Y DB ?
SIZE_X EQU Y-X    ; SIZE_X = 10
```

Вычитание адресов используется обычно для определения расстояния (числа байтов) между этими адресами. В нашем случае разность Y-X показывает, сколько байтов занято всеми элементами массива X.

Отметим одну особенность арифметических операторов: все они выполняются в области 16-битовых чисел, т. е. от результата всегда берутся только последние 16 бит (остаток от деления на 2^{16}, на 10000h), например:

```
(2*9000h)/100h = (12000h)/100h -> 2000h/100h = 20h (а не 120h)
```

2.5.2. Адресные выражения

Значениями адресных выражений являются 16-битовые адреса. Все операции над адресами выполняются по модулю 2^{16} (10000h).

К простейшим адресным выражениям относятся:

- метка (имя команды) и имя переменной, описанное в директиве DB, DW или DD (значениями таких выражений являются адреса меток и имен);
- счетчик размещения; он записывается как $ и обозначает адрес того предложения, в котором он встретился. При трансляции программы ассемблер следит за тем, в ячейку с каким адресом должен попасть машинный эквивалент очередного предложения программы на ЯА; так вот, если мы хотим сослаться на этот адрес, то и надо указать $. Отсюда следует, что в разных предложениях $ обозначает разные адреса. Например, если адрес переменной A равен 100h, то имеем

```
A DW $     ; эквивалентно A DW 100h
B DW $     ; эквивалентно B DW 102h
```

Чаще всего счетчик размещения используется для вычисления размера памяти, занимаемой каким-то массивом, например:

```
    X DW 40 DUP(?)
SIZE_X EQU $-X    ; SIZE_X = 80
```

Здесь $ обозначает адрес первого байта за массивом X, из этого адреса и вычитается начальный адрес массива.

Среди операторов, значением которых являются адреса, пока отметим лишь операторы сложения и вычитания (a – адресное выражение, k – константное):

- сложение адреса с числом: a+k, k+a (значением такого выражения является адрес, полученный прибавлением к адресу a числа k).
- вычитание константы из адреса: a-k (значением является адрес a, уменьшенный на величину k).

Адресные выражения вида <имя>±<целое> используются для ссылок на безымянные ячейки памяти (как правило, элементов массивов). Например, если имеются описания

```
A DB 0,1,2,3
B DB 4
```

то именованными оказываются только байты со значениями 0 и 4, а остальные элементы массива A оказываются безымянными, поэтому к ним нельзя обратиться по имени. Для доступа к таким безымянным ячейкам в ЯА и используются адресные выражения указанного вида. Например, сослаться на байт со значением 3 можно с помощью выражения A+3 или B-1.

Отметим, что в ЯА запрещено вычитание адреса из числа, не допускается сложение, умножение и деление адресов – все это бессмысленные операции. Однако допускается вычитание адресов, но в этом случае, как мы уже знаем, результатом будет не адрес, а константа.

И еще одно замечание. В ЯА указывать адреса в явном, цифровом, виде нельзя. Если указано число, то оно всегда воспринимается как константа, а не адрес. Если же все-таки надо указать адрес в явном виде, то это делается специальным образом, о чем будет рассказано в гл. 7.

3. ПЕРЕСЫЛКИ. АРИФМЕТИЧЕСКИЕ КОМАНДЫ

Мы переходим к изучению команд ПК и способов их записи в ЯА. В данной главе будут рассмотрены команды пересылки и арифметические команды.

3.1. Обозначение операндов команд

При описании команд нам придется указывать, какие операнды в них допустимы, а какие – нет. Для сокращения подобного рода указаний мы введем сейчас обозначения, которыми затем будем пользоваться при описании команд.

Местонахождение операнда	*Обозначение*	*Запись в ЯА*
Команда	i8, i16, i32	Константное выражение
Регистр общего назначения	r8, r16	Имя регистра
Сегментный регистр	sr	CS, DS, SS, ES
Ячейка памяти	m8, m16, m32	Адресное выражение

Непосредственные операнды (т. е. задаваемые в самой команде) мы будем обозначать буквой i (от immediate, непосредственный), указывая за ней, сколько разрядов – 8 (байт), 16 (слово) или 32 (двойное слово) – отводится на него в команде. При этом отметим: запись i16 не означает, что операнд не может быть небольшим числом, в качестве i16 можно указать и операнд 0, и операнд 32000, лишь бы он занимал не более 16 разрядов. В ЯА непосредственные операнды записываются в виде константных выражений.

Регистры будем обозначать буквой r (от register), указывая за ней требуемый размер регистра: r8 – это байтовые регистры (AH, AL, BH и т. п.), а r16 – регистры размером в слово (AX, BX, SI и т. п.). При этом буквой r мы обозначаем только регистры общего назначения, сегментные же регистры мы будем обозначать как sr.

Если операнд находится в памяти, то в команде указывается адрес соответствующей ячейки. Такие операнды мы обозначаем буквой m (от memory, память) с указанием размера ячейки. В ЯА эти операнды задаются адресными выражениями.

3.2. Команды пересылки

В ПК достаточно много команд пересылки, здесь мы рассмотрим только две из них – MOV и XCHG. Кроме того, рассмотрим оператор PTR.

3.2.1. Команда MOV

В ПК есть машинные команды пересылки байта или слова (переслать одной командой двойное слово нельзя). Пересылаемая величина берется из команды, регистра или ячейки памяти, а записывается в регистр или ячейку памяти (записывать в команду, естественно, нельзя). Таких команд много, но в ЯА все они записываются одинаково:

```
Пересылка (move): MOV op1,op2
```

Поэтому при программировании на ЯА можно считать, что имеется только одна команда пересылки. Ассемблер, проанализировав указанные в этой символьной команде операнды, сам выберет нужную машинную команду пересылки, которую и запишет в формируемую им машинную программу.

По команде MOV на место первого операнда пересылается значение второго операнда: op1:=op2. Флаги эта команда не меняет.

Примеры:

```
MOV AX,500    ; A:=500
MOV BL,DH     ; BL:=DH
```

В команде MOV допустимы следующие комбинации операндов:

op1	*op2*	
r8	i8, r8, m8	пересылка байтов
m8	i8, r8	
r16	i16, r16, sr, m16	пересылка слов
sr (кроме CS)	r16, m16	
m16	i16, r16, sr	

Из этой таблицы видно, что запрещены пересылки из одной ячейки памяти в другую, из одного сегментного регистра в другой, запись непосредственного операнда в сегментный регистр. Это обусловлено тем, что в ПК нет таких машинных команд. Если по алгоритму все же нужно такое действие, то оно реализуется в две команды, пересылкой через какой-нибудь несегментный регистр. Например, записать число 100 в сегментный регистр DS можно так:

```
MOV AX,100
MOV DS,AX  ; DS:=100
```

Отметим также, что командой MOV нельзя менять содержимое сегментного регистра CS. Это связано с тем, что регистры CS и IP определяют адрес той команды программы, которая должна быть выполнена следующей, поэтому изменение любого из этих регистров есть ничто иное, как операция перехода, а не пересылка. Команда же MOV не реализует переход.

Как известно, в ПК числа размером в слово хранятся в памяти в "перевернутом" виде, а в регистрах – в нормальном, неперевернутом. Команда MOV учитывает это и при пересылке слов между памятью и регистрами сама "переворачивает" их:

```
Q DW 1234h ; Q: 34h, Q+1: 12h
  ...
  MOV AX,Q ; AH=12h, AL=34h
```

По команде MOV можно переслать как байт, так и слово. А как узнать, что именно – байт или слово – пересылает команда? Не может ли получиться так, что мы хотели написать команду пересылки слова, а оказалось, что команда пересылает байт? Ответ такой: размер пересылаемой величины опре-

деляется по типу операндов, указанных в символьной команде MOV. Более точно ситуация здесь следующая.

Пусть, к примеру, имеются такие описания переменных:

```
X DB ?     ; TYPE X = BYTE
Y DW ?     : TYPE Y = WORD
```

Как правило, в команде MOV легко узнается тип (размер) одного из операндов, он и определяет размер пересылаемой величины. Например:

```
MOV BH,0 ; пересылка байта (BH – байтовый регистр)
MOV X,0  ; то же самое (X описан как имя байтовой переменной)
MOV SI,0 ; пересылка слова (SI – регистр размером в слово)
MOV Y,0  ; то же самое (Y описан как имя переменной-слова)
```

Отметим, что здесь по второму операнду (0) нельзя определить, какого он размера: ноль может быть и байтом (00h), и словом (0000h).

Если можно определить размеры обоих операндов, тогда эти размеры должны совпадать (либо байты, либо слова), иначе ассемблер зафиксирует ошибку. Например:

```
MOV DI,ES   ; пересылка слова
MOV CH,X    ; пересылка байта
MOV DX,AL   ; ошибка (DX – слово, AL – байт)
MOV BH,300  ; ошибка (BH – байт, а 300 не может быть байтом)
```

Отметим, что при пересылках никаких преобразований байта в слово или слова в байт не производится.

3.2.2. Оператор указания типа (PTR)

А теперь рассмотрим ситуацию, когда по операндам команды MOV нельзя определить размер пересылаемой величины.

Забегая вперед, отметим, что если некоторый адрес A надо модифицировать, скажем, по регистру SI, то в ЯА это записывается так: A[SI]. Исполнительный адрес в этом случае вычисляется по формуле Аисп=A+[SI]. В частности, при A=0 эта запись имеет вид [SI] (0 не указывается) и задает исполнительный адрес, равный содержимому регистра SI: Аисп=[SI].

С учетом этого рассмотрим такую задачу. Пусть в регистре SI находится адрес некоторой ячейки памяти и требуется записать 0 в эту ячейку. Тогда, казалось бы, такое обнуление можно сделать с помощью команды

```
MOV [SI],0
```

Однако это не так. Дело в том, что по этой команде нельзя понять, какого размера ноль пересылается, поскольку второй операнд может обозначать ноль как размером в байт (00h), так и размером в слово (0000h), а что касается первого операнда, то адрес из регистра SI может быть адресом ячейки как размером в байт, так и размером в слово (напомним, что с одного и того же адреса могут начинаться ячейки разных размеров).

Итак, в этой команде ни по первому, ни по второму операнду нельзя определить размер пересылаемой величины, а потому ассемблер не сможет определить, на какую конкретную машинную команду заменять эту символьную команду. В подобных ситуациях ассемблер фиксирует ошибку, сообщая, что типы операндов неизвестны. Чтобы не было этой ошибки, автор программы должен явно указать тип хотя бы одного из операндов команды.

Для этого в ЯА введен оператор указания типа PTR (от pointer, указатель), который записывается следующим образом:

```
<тип> PTR <выражение>
```

где <тип> – это BYTE, WORD или DWORD (есть и другие варианты, но мы их пока не рассматриваем), а выражение может быть константным или адресным.

Если указано константное выражение, то оператор "говорит", что значение этого выражения (число) должно рассматриваться ассемблером как величина указанного типа (размера); например, BYTE PTR 0 – это ноль как байт, а WORD PTR 0 – это ноль как слово (запись BYTE PTR 300 ошибочна, т. к. число 300 не может быть байтом). Отметим, что в этом случае оператор PTR относится к константным выражениям.

Если же в PTR указано адресное выражение, то оператор "говорит", что адрес, являющийся значением выражения, должен восприниматься ассемблером как адрес ячейки указанного типа (размера); например: WORD PTR A – адрес A обозначает слово (байты с адресами A и A+1). В данном случае оператор PTR относится к адресным выражениям.

С использованием оператора PTR наша задача решается так: если мы имеем в виду обнуление байта по адресу из регистра SI, то для этого надо использовать команду

```
MOV BYTE PTR [SI],0  или  MOV [SI], BYTE PTR 0
```

а если надо переслать нулевое слово, то команду

```
MOV WORD PTR [SI],0  или  MOV [SI], WORD PTR 0
```

Отметим, что обычно принято уточнять тип операнда-адреса, а не тип непосредственного операнда.

Оператор PTR полезен еще в одной ситуации – когда надо не уточнить тип операнда, а изменить его. Пусть, к примеру, Z – переменная размером в слово:

```
Z DW 1234h ;Z: 34h, Z+1: 12h
```

и надо записать ноль не во все это слово, а только в его первый байт – в тот, где находится величина 34h. Так вот, сделать это командой

```
MOV Z,0
```

нельзя, т. к. по ней 0 запишется в оба байта. Почему? Имя Z описано в директиве DW и потому, как мы уже знаем, получает тип WORD: TYPE Z = = WORD. Когда ассемблер определяет размер операнда команды, в качестве которого указано имя переменной, то он учитывает тип, полученный именем

при описании. Поэтому в нашем случае ассемблер и считает, что пересылается нулевое слово. Обычно так и должно быть, но сейчас нас это не устраивает, нам сейчас нужно, чтобы ассемблер рассматривал Z как имя байта. Вот такое изменение типа имени и позволяет сделать оператор PTR:

```
MOV BYTE PTR Z,0 ; Z: 00h, Z+1: 12h
```

Здесь мы сказали ассемблеру, чтобы он игнорировал тот тип имени Z, который был приписан ему при описании, и считал, что имя Z обозначает байт. (Отметим, что такое изменение типа локально, оно действует только в данной команде.)

Аналогичная ситуация возникает, если мы хотим получить доступ ко второму байту переменной Z. Например, для записи 15 в этот байт нельзя использовать команду

```
MOV Z+1,15
```

т. к. считается, что адрес Z+1 обозначает слово. Это общее правило в ЯА: адрес вида <имя>±<целое> имеет тот же тип, что и <имя>. Поэтому число 15 запишется в два байта – с адресами Z+1 и Z+2. Если же мы хотим изменить только байт по адресу Z+1, тогда надо воспользоваться оператором PTR:

```
MOV BYTE PTR (Z+1),15
```

Здесь конструкция BYTE PTR (Z+1) "говорит", что Z+1 надо рассматривать как адрес байта, а не слова.

Отметим, что в ЯА оператор PTR по старшинству выполняется до оператора сложения, поэтому запись BYTE PTR Z+1 трактуется как (BYTE PTR Z)+1. Однако в данном конкретном случае старшинство операторов не играет никакой роли, т. к. обе записи – BYTE PTR (Z+1) и BYTE PTR Z+1 – эквивалентны по смыслу: в первом случае мы сначала увеличиваем адрес Z на 1 и только затем сообщаем, что Z+1 надо рассматривать как адрес байта, а во втором случае мы сначала сообщаем, что Z – это адрес байта, и лишь затем увеличиваем его на 1 (при этом "байтовость" адреса сохраняется).

Итак, оператор PTR используется в следующих ситуациях: когда типы операндов команд неизвестны и потому надо указать явно тип одного из операндов, и когда нас не устраивает тип, приписанный имени при его описании, и потому мы должны указать нужный нам тип.

3.2.3. Команда XCHG

В машинных программах приходится довольно часто переставлять местами какие-то две величины, и хотя такую перестановку можно реализовать только с помощью команды MOV, в ПК введена специальная команда для этого:

```
Перестановка (exchange): XCHG op1,op2
```

Эта команда меняет местами значения своих операндов (они должны быть либо байтами, либо словами): op1 <=> op2. Флаги при этом не меняются.

Пример:

```
MOV AX,62     ;AX=62
MOV SI,135    ;SI=135
XCHG AX,SI    ;AX=135, SI=62
```

Допустимые типы операндов команды XCHG:

op1	*op2*	
r8	r8, m8	перестановка байтов
m8	r8	
r16	r16, m16	перестановка слов
m16	r16	

Как видно, не допускается перестановка содержимого двух ячеек памяти. Если все-таки надо сделать такую перестановку, то это реализуется через какой-нибудь регистр. Например, поменять местами значения байтовых переменных X и Y можно так:

```
MOV AL,X   ;AL=X
XCHG AL,Y  ;AL=Y, Y=X
MOV X,AL   ;X=Y (исходное значение)
```

3.3. Команды сложения и вычитания

Прежде чем перечислить имеющиеся в ПК команды сложения и вычитания, рассмотрим особенности выполнения этих операций.

3.3.1. Особенности сложения и вычитания целых чисел в ПК

Беззнаковые числа складываются как обычно, только в двоичной системе счисления. Однако здесь есть одна проблема: что делать, если сумма получится очень большой – такой, что она не умещается в ячейке? Например, если при ячейке размером в 8 бит мы складываем 250 и 10, то получим число 260 (100000100b), которое не "влезает" в ячейку.

В некоторых ЭВМ в такой ситуации фиксируется ошибка и на этом прекращается выполнение программы. Однако в ПК реакция иная: ошибка не фиксируется, левая единица (единица переноса) отбрасывается и в качестве ответа выдается искаженная сумма (в нашем примере ответом будет байт 00000100b, т. е. число 4). Но зато в флаг переноса CF записывают 1. Это сигнал о том, что получилась неправильная сумма (если переноса не было, то в CF записывают 0). Затем можно проанализировать этот флаг (подходящие средства для этого есть) и "поймать" такую ошибку.

Такое суммирование с отбрасыванием единицы переноса в математике называется суммированием по модулю 2^k (k – размер ячейки), при этом в флаге CF фиксируется, был ли перенос:

$$\text{сумма}(x,y)=(x+y) \bmod 2^k=\begin{cases} x+y, & \text{если } x+y<2^k, \ CF=0 \\ x+y-2^k, & \text{если } x+y\geq 2^k, \ CF=1 \end{cases}$$

При вычитании беззнаковых целых чисел также возникает проблема: что делать, если при вычитании x-y число x меньше числа y? Ведь в этом случае получится отрицательная разность, а это уже вне области беззнаковых чисел.

В ПК поступают следующим образом: при x>=y выполняется обычное вычитание, но если x<y, тогда числу x дается "заем" единицы (к числу x прибавляется величина 2^k) и только после этого производится вычитание. Полученное таким образом число и объявляется разностью. Например, при k=8 вычитание 1–2 происходит таким образом:

```
1-2 ==> (2^8+1)-2 = (256+1)-2 = 257-2 = 255
```

(в двоичной системе замена 1 на 256+1 – это замена 00000001 на 100000001, т. е. приписывание 1 слева) и именно число 255 объявляется результатом вычитания 1-2. При этом ошибка не фиксируется, зато в флаг переноса CF заносится 1, что сигнализирует о заеме единицы, о неправильном результате (при x>=y в CF заносится 0).

Выполняемое так вычитание в математике называют вычитанием по модулю 2^k, при этом фиксируется, был ли заем:

$$\text{разность}(x,y)=(x-y) \bmod 2^k=\begin{cases} x-y, & \text{если } x\geq y, \ CF=0 \\ (2^k+x)-y, & \text{если } x<y, \ CF=1 \end{cases}$$

Итак, в ПК сложение и вычитание беззнаковых целых чисел – это на самом деле сложение и вычитание по модулю 2^k, где k – размер ячеек. Причем появление после операции значения 1 в флаге переноса CF свидетельствует о том, что выданный ответ неправильный.

Теперь рассмотрим сложение и вычитание знаковых чисел. Оказывается, если целые со знаком представлены в дополнительном коде, то складывать и вычитать их можно по алгоритмам для беззнаковых чисел. Делается это так: дополнительные коды знаковых операндов рассматривают как числа без знака и в таком виде их складывают или вычитают, а полученный результат затем рассматривают как дополнительный код знакового ответа.

Пример (при ячейке в 8 бит). Пусть надо сложить +3 и -1. Их дополнительные коды – это 3 и (256-1)=255. Складываем их как числа без знака: 3+255 (mod 256) = 258 (mod 256) = 2. Теперь величина 2 рассматривается как дополнительный код ответа, поэтому получается ответ +2.

Другой пример. Пусть надо сложить -3 и +1. Дополнительные коды этих знаковых чисел: (256-3)=253 и 1. Складываем их как беззнаковые числа: 253+1 (mod 256) = 254. Теперь, рассматривая эту величину как дополнительный код ответа, получаем результат -2 (254=256-2).

Из сказанного следует, что в ЭВМ, где знаковые числа представляются в дополнительном коде, не нужны разные машинные команды для сложения и вычитания беззнаковых и знаковых чисел, достаточно и одного набора этих команд. (В этом важное преимущество дополнительного кода над другими способами представления знаковых чисел.)

Однако не все так просто при сложении и вычитании знаковых чисел, здесь есть свои неприятности. Напомним, что при размере ячеек в 8 бит в дополнительном коде представляются только числа от -128 до +127. Рассмотрим, к примеру, сложение знаковых чисел +127 и +2. Складывая их как беззнаковые числа 127 и 2, получаем величину 129, которую теперь надо рассмотреть как дополнительный код ответа: поскольку 129=256-127, то суммой должно быть признано число -127. Таким образом, складывая два положительных числа, мы получили отрицательное число!

Почему так произошло? При представлении чисел (размером в байт) в дополнительном коде левый разряд является знаковым, а на модуль числа отводится 7 правых разрядов. У нас же получился ответ 129=10000001b, модуль которого не вмещается в эти 7 разрядов, поэтому модуль и "залез" в знаковый разряд, изменив его на противоположный.

Такое налезание модуля (мантиссы, цифровой части) числа на знаковый разряд называют "переполнением мантиссы". В общем случае оно происходит, если складываются числа одного знака и настоящая сумма оказывается вне диапазона представимых знаковых чисел ([-128, +127] при k=8). Переполнение мантиссы фиксируется в флаге переполнения OF: он получает значение 1, если было переполнение, и значение 0 иначе. Таким образом, при OF=0 результат правильный, а при OF=1 – неправильный, однако эта ошибка не фиксируется и "поймать" ее можно только последующим анализом флага OF.

Аналогичное переполнение мантиссы возможно и при вычитании. Например, при вычитании (+127)-(-2) = 127+2 получаем 129, а это дополнительный код числа -127, которое и выдается как результат вычитания, хотя истинной разностью является число 129. В общем случае переполнение мантиссы происходит, если вычитаются числа разных знаков и настоящая разность оказалась вне диапазона представимых знаковых чисел. И здесь факт переполнения фиксируется в флаге OF: он получает значение 1, если было переполнение мантиссы и результат операции неправильный, и значение 0, если не было переполнения и ответ правильный.

Итак, при сложении и вычитании как беззнаковых, так и знаковых чисел возможны особые случаи, когда настоящий (в математическом смысле) результат выходит за диапазон представимых чисел, и тогда результат искажа-

ется. Такое искажение результата фиксируется в флагах CF и OF. Распознать такую ошибку можно лишь последующим анализом этих флагов.

И еще одно замечание. Поскольку сложение и вычитание беззнаковых и знаковых чисел производятся по одним и тем же алгоритмам и поскольку ПК заранее не знает, какие именно числа он складывает или вычитает, то при выполнении этих операций ПК одновременно фиксирует в флагах CF и OF особенности операций для обоих классов чисел. Какие именно числа складываются (вычитаются), знает только автор программы, он и должен решать, на какой из этих двух флагов следует реагировать. Если он считает, что складываются беззнаковые числа, то для него представляет интерес флаг CF (был ли перенос) и безразличен флаг OF, но если, по его мнению, складываются знаковые числа, то он должен интересоваться флагом OF (было ли переполнение мантиссы) и не должен обращать внимание на флаг CF.

При сложении и вычитании чисел меняются также флаг нуля ZF и флаг знака SF. Флаг ZF получает значение 1, если результат оказался нулевым, и значение 0, если результат ненулевой; этот флаг представляет интерес при работе как со знаковыми, так и беззнаковыми числами. В флаг же SF заносится знаковый (самый левый) бит результата; этот флаг полезен при работе со знаковыми числами, т. к. он получает значение 1, если результат оказался отрицательным, и значение 0 иначе.

Примеры (ячейки размером в байт):

```
9-9 = 0 = 00000000b ==> ZF=1, SF=0
8-9 = -1 = 11111111b ==> ZF=0, SF=1
9-8 = 1 = 00000001b ==> ZF=0, SF=0
```

3.3.2. Команды сложения и вычитания

В ПК имеется несколько команд сложения и вычитания. Основными из них являются следующие:

```
Сложение:                ADD op1,op2
Вычитание (subtract):    SUB op1,op2
```

В этих командах допустимы следующие комбинации операндов:

op1	*op2*	
r8	i8, r8, m8	сложение/вычитание байтов
m8	i8, r8	
r16	i16, r16, m16	сложение/вычитание слов
m16	i16, r16	

Команда ADD складывает операнды и записывает их сумму на место первого операнда: op1:=op1+op2. По команде SUB из первого операнда вычитается второй операнд и полученная разность записывается вместо первого операнда: op1:=op1-op2.

Например:

```
ADD AH,12     ;AH:=AH+12
SUB SI,Z      ;SI:=SI-Z
ADD Z,-300    ;Z:=Z+(-300)
```

Мы уже рассмотрели особенности сложения и вычитания целых чисел, поэтому отметим лишь следующее. Команды ADD и SUB работают как с числами размером в байт, так и с числами размером в слово; нельзя, чтобы один операнд был байтом, а другой – словом. При этом числа могут быть как знаковыми, так и беззнаковыми. В этих командах меняются флаги переноса CF, переполнения OF, знака SF и нуля ZF (а также флаги AF и PF, но мы их не будем рассматривать), правила изменения которых мы уже рассмотрели.

Следующая пара команд сложения и вычитания:

```
Увеличение на 1 (increment): INC op
Уменьшение на 1 (decrement): DEC op
```

В этих командах допустимы следующие типы операнда: r8, m8, r16, m16.

Примеры:

```
INC BL
DEC WORD PTR A
```

Команда INC аналогична команде ADD op,1, т. е. увеличивает свой операнд на 1: op1:=op1+1, а команда DEC аналогична команде SUB op,1, т. е. уменьшает операнд на 1: op1:=op1-1 (единственное отличие: команды INC и DEC не меняют флаг переноса CF). Выгода от команд INC и DEC в том, что они занимают меньше места в памяти и выполняются быстрее, чем соответствующие команды ADD и SUB.

Еще одна команда из группы сложения и вычитания:

```
Изменение знака (negative): NEG op
```

Допустимые типы операнда этой команды: r8, m8, r16, m16.

Команда NEG рассматривает свой операнд как число со знаком и меняет его знак на противоположный: op:=-op. Например:

```
MOV AH,1
NEG AH     ;AH:=-1 (0FFh)
```

Здесь есть особый случай: если op=-128 (80h), то операнд не меняется, т. к. нет знакового числа +128. Аналогично для чисел-слов: если значение операнда равно минимальному отрицательному числу -32768 (8000h), то команда не меняет операнд.

В этом особом случае флаг OF получает значение 1 (при других операндах OF=0). При нулевом операнде флаг CF равен 0, при других – 1. Флаги SF и ZF меняются как обычно.

И, наконец, рассмотрим еще пару команд сложения и вычитания:

```
Сложение с учетом переноса (add with carry):      ADC op1,op2
Вычитание с учетом заема (subtract with borrow):  SBB op1,op2
```

Допустимые типы операндов – как в командах ADD и SUB.

Эти команды аналогичны командам обычного сложения и вычитания (ADD и SUB) за одним исключением: в команде ADC к сумме операндов еще прибавляется значение флага переноса CF: op1:=op1+op2+CF, а в команде SBB из разности операндов еще вычитается значение этого флага: op1:=op1-op2-CF.

Зачем это нужно? В ПК одной командой можно сложить (вычесть) только числа размером в байт или слово. Сложение (вычитание) чисел других размеров, например двойных слов, приходится реализовывать нам самим через сложение (вычитание) чисел размером в слово или байт. Здесь-то и оказываются полезными команды ADC и SBB.

Рассмотрим, для примера, как можно сложить следующие два числа размером в двойное слово: X=1204F003h и Y=8052300Fh. Условно разбиваем каждое число на два слова. Сначала складываем младшие (правые) части их, используя команду ADD. Может получиться единица переноса, которую надо учесть при сложении старших (левых) частей чисел. Как это сделать? Вспомним, что единица переноса попадает в флаг CF, поэтому к сумме старших частей надо добавить и значение этого флага (если единицы переноса не было, то CF=0, поэтому и здесь можно прибавлять CF), а такое сложение как раз и осуществляет команда ADC. Следовательно, старшие части чисел надо складывать по команде ADC.

```
X = 1204    F003
  +       +
Y = 8052    300F
    9256    12012
  +         ↓
      1 ←CF
    9257    2012
```

Если для определенности считать, что число X размещается в двух регистрах AX (старшие цифры) и BX (младшие), а число Y – в регистрах CX (старшие цифры) и DX (младшие), и если сумму этих двух чисел надо записать вместо числа X, т. е. надо реализовать (AX,BX):=(AX,BX)+(CX,DX), тогда это делается так:

```
ADD BX,DX  ;BX:=Хмл+Умл, CF=перенос
ADC AX,CX  ;AX:=Хст+Уст+CF
```

(Отметим, что при сложении старших частей также может появиться единица переноса, однако мы ее уже не будем учитывать.)

Аналогичным образом реализуется вычитание беззнаковых чисел размером в двойное слово, для чего используется команда SBB. Например, вычитание (AX,BX):=(AX,BX)-(CX,DX) реализуется так:

```
SUB BX,DX  ;BX:=Хмл-Умл, CF=заем единицы
SBB AX,CX  ;AX:=Хст-Уст-CF
```

С помощью команд ADC и SBB можно реализовать сложение и вычитание чисел любого размера, причем эти операции для беззнаковых и знаковых чисел реализуются одинаково.

3.4. Команды умножения и деления

3.4.1. Команды умножения

Если сложение и вычитание беззнаковых и знаковых чисел производятся по одним и тем же алгоритмам, то умножение чисел этих двух классов выполняется по разным алгоритмам, в связи с чем в ПК имеются две команды умножения:

```
Умножение целых без знака (multiply):          MUL op
Умножение целых со знаком (integer multiply): IMUL op
```

В остальном эти команды действуют одинаково:

```
Умножение байтов: AX:=AL*op        (op: r8, m8)
Умножение слов:  (DX,AX):=AX*op    (op: r16, m16)
```

Операнд op, указываемый в команде, – это лишь один из сомножителей; он может находиться в регистре или в памяти, но не может быть непосредственным операндом. Местонахождение другого сомножителя фиксировано и потому в команде не указывается явно: при умножении байтов он берется из регистра AL, а при умножении слов – из регистра AX.

Местонахождение результата умножения также заранее известно и потому в команде явно не указывается. При этом под результат отводится в два раза больше места, чем под сомножители. Это связано с тем, что умножение n-значных чисел в общем случае дает произведение из 2n цифр, и с желанием сохранить все цифры произведения. При умножении байтов результат имеет размер слова и записывается в весь регистр AX (в AH – старшие цифры произведения, в AL – младшие), а при умножении слов результат имеет размер двойного слова и записывается в два регистра – в регистр DX заносятся старшие цифры произведения, а в регистр AX – младшие цифры.

Примеры:

```
N DB 10
  ...
  MOV AL,2
  MUL N       ; AX=2*10=20=0014h: AH=00h, AL=14h
  MOV AL,26
  MUL N       ; AX=26*10=260=0104h: AH=01h, AL=04h
  MOV AX,8
  MOV BX,-1
  IMUL BX     ; (DX,AX)=-8=0FFFFFFF8h: DX=0FFFFh, AX=0FFF8h
```

Итак, команды умножения выдают результат в удвоенном формате. Это не всегда удобно: то мы работали с числами-байтами, а тут приходится переходить на обработку чисел-слов. В то же время далеко не всегда величина произведения столь велика, что ему нужен удвоенный формат; например,

в первой и третьей из наших команд умножения для результата вполне достаточно было бы обычного формата. Поэтому важно знать, действительно ли произведению нужен двойной формат или ему достаточно и одинарного формата. Иногда об этом известно заранее (мы заранее знаем, что перемножаются небольшие числа), но иногда это можно установить только после умножения. В последнем случае вопрос о том, умещается ли результат умножения в формат сомножителей или нет, решается с помощью анализа флагов переноса CF и переполнения OF, которые в обеих командах умножения меняются синхронно и по следующему правилу:

CF=OF=1 – если произведение занимает двойной формат
CF=OF=0 – если произведению достаточен формат сомножителей

При CF=0 (одновременно и OF=0) можно считать, что произведение байтов занимает только регистр AL, а произведение слов – только регистр AX, и дальше можно работать только с этими регистрами. Но если CF=1, то далее приходится работать с произведением как с числом удвоенного формата.

3.4.2. О команде умножения в процессорах 80186 и старше

Фиксированность местонахождения первого операнда и результата и невозможность указывать непосредственный операнд в командах умножения делают эти команды не очень удобными. Поэтому в процессоре 80186 была введена новая команда умножения с тремя операндами, где некоторые из этих неудобств устранены. В ЯА эта команда допускает две эквивалентные формы записи:

```
MUL op1,op2,op3    или     IMUL op1,op2,op3
```

и реализует следующее действие: op1:=op2*op3.

Допустимые типы операндов: op1: r16; op2: r16, m16; op3: i16.

Таким образом, результат умножения можно поместить в любой регистр общего назначения, первый сомножитель может находиться как в регистре общего назначения, так и в любом слове памяти, а второй сомножитель обязательно задается как непосредственный операнд. Например:

```
X DW ?
  ...
  MUL SI,BX,3;SI:=BX*3
  IMUL DX,X,-10 ;DX:=X*(-10)
```

Данная команда умножения предназначена для умножения только чисел размером в слово и только при условии, что произведение умещается в слово. При этом условии нет разницы между умножением чисел со знаком и чисел без знака, поэтому-то в ЯА и можно использовать любое из двух названий этой команды. Если указанное условие действительно выполняется, тогда флаги CF и OF получают значение 0, но если результат превосходит размер слова, тогда левые, наиболее значимые, биты произведения теряются, в регистр op1 записываются последние 16 бит, а флаги CF и OF устанавливаются в 1.

Допускается следующее сокращение в записи рассматриваемой команды умножения: если op1=op2, т. е. если первый сомножитель берется из регистра, в который должен быть записан результат, тогда в записи команды этот регистр может быть указан только раз:

```
MUL op1,op3 (IMUL op1,op3) – сокращение для MUL op1,op1,op3
```

Например:

```
MUL DX,-15  ;DX:=DX*(-15)
```

Важное замечание. По умолчанию ассемблер разрешает указывать в тексте программы на ЯА только те команды, которые имеются в процессоре 8086, и фиксирует ошибку, если указаны команды, которые появились лишь в более старших моделях процессоров семейства 80x86. Так вот, чтобы ассемблер правильно воспринимал новые команды, скажем, процессора 80186 (в частности, новую команду умножения в любом ее варианте), в текст программы (в любом месте, но до первой такой команды) должна быть помещена директива

```
.186
```

разрешающая до конца текста программы или до следующей подобной директивы (об этих директивах см. разд. 14.6) указывать все команды процессора 80186, которые можно использовать и в процессорах старших моделей.

Пример. Пусть X – байтовая переменная, а Y – переменная размером в слово и надо выполнить присваивание Y:=5*X*X-7*Y при условии, что числа знаковые и все вычисления укладываются в размер слова. Тогда это можно сделать так:

```
.186            ;допускаются все команды процессора 80186
...
MOV AL,X
IMUL AL         ;AX:=X*X    ("обычное" умножение)
IMUL AX,5       ;AX:=X*X*5 (можно: MUL AX,5 или IMUL AX,AX,5)
IMUL BX,Y,-7    ;BX:=Y*(-7)
ADD AX,BX
MOV Y,AX
```

В данной книге мы не собираемся выходить за рамки системы команд процессора 8086, поэтому в дальнейшем мы не будем пользоваться рассмотренным здесь вариантом команды умножения.

3.4.3. Команды деления

Как и умножение, деление чисел без знака и со знаком также реализуется двумя командами:

```
Деление целых без знака (divide):          DIV op
Деление целых со знаком (integer divide): IDIV op
```

Первая из этих команд предназначена для деления беззнаковых целых чисел, а вторая – для деления знаковых чисел, в остальном же эти команды действуют одинаково:

- деление слова на байт:

```
AH:=AX mod op, AL:=AX div op            (op: r8, m8)
```

- деление двойного слова на слово:

```
DX:=(DX,AX) mod op, AX:=(DX,AX) div op  (op: r16, m16)
```

Как видно, в этих командах местонахождение первого операнда (делимого) и результата фиксировано и потому явно не указывается. Указывается только второй операнд (делитель), который может находиться в регистре или в ячейке памяти, но не может быть непосредственным операндом.

При делении слова на байт делимое обязано находиться в регистре AX, а делитель должен быть байтом. При делении двойного слова на слово делимое обязано находиться в двух регистрах – в DX (старшая часть делимого) и в AX (младшая часть), а делитель должен иметь размер слова.

В области целых чисел "настоящее" деление невозможно, и в ПК под делением понимают получение сразу двух величин – неполного частного (div) и остатка (mod). Оба этих числа помещаются на место делимого: его старшая часть заменяется на остаток, а младшая – на неполное частное. Оба этих числа имеют один и тот же размер, совпадающий с размером второго операнда (делителя).

Если через функцию trunc(x) обозначить отбрасывание дробной части вещественного числа x, тогда операции div и mod определяются следующим образом:

```
a div b = trunc(a/b)
a mod b = a-b*(a div b)
```

Примеры:

```
13 div 4 = trunc(3.25) = 3          13 mod 4 = 13-4*3 = 1
(-13) div 4 = trunc(-3.25) = -3     (-13) mod 4 = (-13)-4*(-3) = -1
13 div (-4) = trunc(-3.25) = -3     13 mod (-4) = 13-(-4)*(-3) = 1
(-13) div (-4) = trunc(3.25) = 3    (-13) mod (-4) = (-13)-(-4)*3 = -1
```

И, наконец, отметим, что при выполнении команды деления возможно появление ошибки с названием "деление на 0 или переполнение". Она возникает в двух случаях:

- делитель равен 0 (op=0),
- неполное частное не вмещается в отведенное ему место (регистр AL или AX); это, например, произойдет при делении 600 на 2:

```
MOV AX,600
MOV BH,2
DIV BH     ; 600 div 2 = 300, но 300 не вмещается в AL
```

При такой ошибке ПК прекращает выполнение программы.

3.5. Изменение размера числа

Рассмотрим такую задачу. Пусть к числу из регистра BX надо прибавить число из регистра AL: BX:=BX+AL. Здесь требуется сложить слово с байтом, но в ПК нет команды сложения слова с байтом. Поэтому прежде всего надо сделать равными размеры этих чисел, а именно: надо расширить байт AL до слова, записав его, скажем, в регистр AX, и только затем сложить числа из регистров AX и BX. Возникает вопрос: как расширить байт AL до слова AX с сохранением, конечно, величины числа?

Если число трактуется как беззнаковое, то делается это просто – надо слева к числу приписать нули: например, если AL=32=20h, то AX=0020h. Реализуется это записью 0 в левую часть регистра AX, т. е. в регистр AH:

```
MOV AH,0 ; AL --> AX (без знака)
```

Такое расширение называется расширением без знака.

Для чисел со знаком дело осложняется, т. к. надо учитывать знак числа. Если число неотрицательно, то для его расширения достаточно слева приписать нули. Но если число отрицательно, тогда слева надо приписывать двоичные единички (в шестнадцатеричной системе – две цифры F). К примеру, число -32 (-20h) в дополнительном коде как байт имеет вид 256-32 = 100h-20h = = 0E0h, а как слово – 10000h-20h = 0FFE0h.

Итак, расширение знакового числа заключается в приписывании к нему слева нулей, если это число неотрицательно, или единичек, если число отрицательно. Такое расширение называется расширением со знаком.

Этому расширению можно дать и такую трактовку. Вспомним, что в дополнительном коде знакового числа самый левый бит является знаковым и что этот бит равен 0 для неотрицательных чисел и равен 1 для отрицательных. Поэтому можно сказать, что расширение со знаком заключается в дублировании слева знакового бита числа:

```
доп(+32) = 0010 0000b --> 0000 0000 0010 0000b
             +
доп(-32) = 1110 0000b --> 1111 1111 1110 0000b
             -
```

Для осуществления расширения со знаком в систему команд ПК введена специальная команда:

```
Расширение байта до слова (convert byte to word): CBW
```

У этой команды местонахождение операнда и результата фиксировано: операнд всегда берется из AL, а результат всегда записывается в AX. Команда записывает в регистр AH число 00h или 0FFh в зависимости от знака числа из регистра AL:

$$AH:=\begin{cases} 00h & \text{при } AL>=0 \\ 0FFh & \text{при } AL<0 \end{cases}$$

Флаги эта команда не меняет.

Примеры:

```
MOV AL,32     ; AL=20h
CBW           ; AX=0020h (число +32 как слово)
MOV AL,-32    ; AL=0E0h
CBW           ; AX=0FFE0h (число -32 как слово)
```

Теперь вернемся к нашей задаче – к сложению BX:=BX+AL. В зависимости от того, как трактуются числа – как беззнаковые или как знаковые, она решается следующим образом:

```
; числа без знака:
MOV AH,0    ;AL -> AX (расширение без знака)
ADD BX,AX
; числа со знаком:
CBW         ; AL -> AX (расширение со знаком)
ADD BX,AX
```

Необходимость расширения байта до слова чаще всего возникает при делении байтов. Напомним, что в ПК можно делить слово на байт, но не байт на байт, поэтому, если мы хотим все же разделить байт на байт, то нам прежде всего надо первый байт (делимое) расширить до слова. А как это делается, мы только что рассмотрели. Например, присваивание AL:=AL div CH реализуется так:

```
; числа без знака:
MOV AH,0    ; AL -> AX (без знака)
DIV CH      ; AL:=AX div CH (AH:=AX mod CH)
; числа со знаком:
CBW         ; AL -> AX (со знаком)
IDIV CH     ; AL:=A div CH (AH:=AX mod CH)
```

Мы рассмотрели случай расширения байта до слова. Аналогично осуществляется расширение слова до двойного слова, необходимость в котором также возникает при делении. Обычно приходится расширять слово, находящееся в регистре AX, до двойного слова, занимающего два регистра – DX и AX, при условии, что в DX находится старшая часть числа, а в AX – младшая:

```
AX --> (DX,AX)
```

При этом условии расширение беззнакового числа AX реализуется засылкой 0 в регистр DX:

```
MOV DX,0 ; AX -> (DX,AX) (без знака)
```

Если же число AX рассматривается как знаковое, тогда для его расширения надо в регистр DX записать 0000h, если число неотрицательно, или 0FFFFh, если число отрицательно. Для этого в системе команд ПК есть специальная команда:

```
Расширение слова до двойного (convert word to double): CWD
```

Действие этой команды можно описать так:

$$DX:=\begin{cases}0000h & \text{при } AX>=0\\ 0FFFFh & \text{при } AX<0\end{cases}$$

Примеры здесь аналогичны предыдущим.

3.6. Примеры

В этом разделе рассматриваются примеры использования тех команд ПК и средств ЯА, что описаны в данной главе, и приводятся их решения с пояснениями.

Пример 1

```
X DD ?
```

Требуется переставить местами оба слова двойного слова X.

Решение

Какие именно команды здесь нужны, определить несложно. Проблема здесь в другом – в том, что в этих командах нельзя непосредственно указывать адреса X и X+2. Например, при трансляции команды

```
MOV AX,X
```

ассемблер зафиксирует ошибку, т. к. в команде MOV можно указывать операнды только типов BYTE и WORD, а здесь указан операнд X типа DWORD. Чтобы не было такой ошибки, надо явно указать ассемблеру, чтобы он рассматривал X не как имя двойного слова, а как имя слова, для чего следует применить оператор PTR.

С учетом сказанного задача решается так:

```
MOV AX,WORD PTR X      ;AX:=Хлев
XCHG AX,WORD PTR X+2   ;AX:=Хправ, Хправ:=Хлев
MOV WORD PTR X,AX      ;Хлев:=прежнее значение Хправ
```

Воспользуемся этим примером и рассмотрим один из приемов сокращения записи команд. Как видно, в наших командах несколько раз используется конструкция WORD PTR. Так вот, чтобы многократно не выписывать эту достаточно длинную конструкцию, можно с помощью директивы EQU обозначить ее коротким именем, а затем использовать это имя. В результате получим более короткий вариант записи команд:

```
WP EQU WORD PTR
   MOV AX, WP X
   XCHG AX,WP X+2
   MOV WP X,AX
```

Пример 2

```
W DW 1234h
```

Определить значения регистров AX и BX после выполнения команд

```
MOV AX,W
MOV BH,BYTE PTR W
MOV BL,BYTE PTR W+1
```

Решение

Ответ здесь такой: AX=1234h, BX=3412h.

Казалось бы значения регистров AX и BX должны совпадать, а они различаются. Почему? Напомним, что в памяти ПК слова хранятся в "перевернутом" виде; в нашем случае в байте с адресом W находится величина 34h, а в байте с адресом W+1 – величина 12h. Поэтому вторая из приведенных команд записывает в старшую часть регистра BX (в BH) младшие цифры числа W, а в третьей команде в младшую часть регистра BX (в BL) записываются старшие цифры числа, и потому в регистре BX число W оказывается в "перевернутом" виде. Во всех же командах, работающих со словами как с единым целым, такая особенность представления слов в памяти учитывается, и эти команды "переворачивают" слова автоматически. Поэтому первая из наших команд сама "перевернет" слово W и запишет его в регистр AX в нормальном, неперевернутом виде. При использовании таких команд можно забыть о "перевернутом" представлении слов в памяти, но если мы обрабатываем слово по частям (по байтам), тогда автоматического "переворачивания" нет и мы обязаны помнить о "перевернутом" представлении.

Пример 3

Пусть X – байтовая переменная, значение которой трактуется как знаковое число, а Y – переменная размером в слово. Вычислить Y=X*X*X при условии, что результат имеет размер слова.

Решение

При использовании арифметических команд ПК надо внимательно следить за размерами операндов, за правильностью расширения чисел (со знаком или без), а если используются команды умножения, деления и расширения со знаком, то надо следить и за тем, чтобы их операнды находились в нужных регистрах.

В нашем примере мы сначала умножим байт X на себя. Поскольку один из операндов команды IMUL должен находиться в регистре AL, то прежде всего записываем X в этот регистр. Получив в регистре AX произведение X*X, мы должны умножить его на байт X. Поскольку в ПК нельзя умножать слово на байт, то приходится расширять байт X до слова, причем расширять со знаком, для чего воспользуемся командой CBW. Поскольку эта команда использует регистр AX, то предварительно этот регистр надо освободить. Произведение X*X*X уже будет иметь размер двойного слова и будет находиться в регистрах DX и AX. Однако, согласно условию задачи, для нашего произведения достаточно лишь слова, поэтому старшую часть произведения, т. е. содержимое регистра DX, можно проигнорировать.

С учетом всех замечаний получаем такие команды:

```
MOV AL,X
IMUL AL    ;AX:=X*X (можно: IMUL X)
MOV BX,AX  ;спасти AX
MOV AL,X
CBW        ;AX:=X как слово
IMUL BX    ;(DX,AX):=X*X*X (DX можно не учитывать)
MOV Y,AX
```

Пример 4

```
N DB ?
D DB 3 DUP(?)
```

Рассматривая N как беззнаковое число (от 0 до 255), записать в массив D цифры (как символы) из десятичной записи этого числа: в байт D – левую цифру, в байт D+1 – среднюю цифру, в байт D+2 – правую цифру.

Решение

Пусть a, b и c – десятичные цифры числа N: N=abc. Для получения правой цифры c надо взять остаток от деления N на 10. Неполное же частное от такого деления – это число ab, поэтому если его разделить на 10, то неполное частное даст цифру a, а остаток – цифру b. Отметим, что непосредственно делить байт N на 10 нельзя, т. к. в ПК нет команды деления байтов, поэтому предварительно надо расширить N до слова, причем расширить без знака.

Полученные величины a, b и c – это цифры как числа (от 0 до 9). Чтобы получить эти цифры как символы (от '0' от '9'), к ним надо прибавить код символа '0', т. к. в используемых в ПК системах кодировки справедливо следующее соотношение: код(i)=код('0')+i, где i – цифра.

```
MOV BL,10   ;делитель
MOV AL,N
MOV AH,0    ;AX:=N как слово
DIV BL      ;AH:=c, AL:=ab
ADD AH,'0'  ;правая цифра как символ
MOV D+2,AH  ;запомнить c
MOV AH,0    ;AX:=ab как слово
DIV BL      ;AH:=b, AL:=a (как числа)
ADD AX,'00' ;обе цифры как символы (AH:=b+'0', AL:=a+'0')
MOV D+1,AH  ;запомнить b
MOV D,AL    ;запомнить a
```

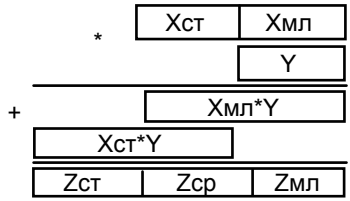
Пример 5

```
X DD ?
Y DW ?
Z DW ?,?,?
```

Вычислить Z=X*Y в предположении, что числа X и Y беззнаковые и что их произведение записывается в тройном слове Z в "перевернутом" виде: младшие (правые) цифры (Zмл) – в слове с адресом Z, средние цифры (Zср) – в слове с адресом Z+2, старшие цифры (Zст) – в слове с адресом Z+4.

Решение

Поскольку в ПК нет команды умножения двойного слова на слово, такое умножение приходится реализовывать нам самим с помощью имеющихся команд. Для этого будем рассматривать двойное слово X как два слова Xст и Xмл и будем выполнять умножение по следующей схеме:



Отметим, что перед адресами Z, Z+2 и Z+4 не надо указывать конструкцию WORD PTR, т. к. имя Z согласно описанию уже имеет тип WORD, а вот перед адресами X и X+2 ее надо ставить.

С учетом всего сказанного получается такое решение задачи:

```
WP EQU WORD PTR
   MOV AX,WP X    ;AX:=Xмл
   MUL Y          ;(DX,AX):=Xмл*Y
   MOV Z,AX       ;Zмл
   MOV BX,DX      ;спасти старшую часть Xмл*Y
   MOV AX,WP X+2  ;AX:=Xст
   MUL Y          ;(DX,AX):=Xст*Y
   ADD AX,BX      ;AX:=(Xст*Y)мл+(Xмл*Y)ст, CF=перенос
   MOV Z+2,AX     ;Zср
   ADC DX,0       ;DX:=(Xст*Y)ст+CF
   MOV Z+4,DX     ;Zст
```

4. ПЕРЕХОДЫ. ЦИКЛЫ

В этой главе рассматриваются команды перехода, способы организации циклов и операции ввода-вывода, которыми мы будем пользоваться на протяжении всей книги.

4.1. Безусловный переход. Оператор SHORT

Команды машинной программы выполняются в том порядке, как они записаны в памяти. Но время от времени этот естественный порядок выполнения команд приходится нарушать с тем, чтобы следующей выполнялась не очередная команда программы, а какая-то иная. Такую возможность обеспечивают команды перехода.

Переходы бывают условными и безусловными. Если переход делается только тогда, когда выполнено некоторое условие, то такой переход называется условным, а если он делается независимо от каких-либо условий, то это безусловный переход.

Отметим, что в ПК команды перехода не меняются флаги: какое значение флаги имели до команды перехода, такое же значение они будут иметь и после нее. Дальше мы уже не будем об этом упоминать.

Изучение команд перехода начнем с безусловного перехода.

В ПК имеется несколько машинных команд безусловного перехода, но в ЯА они все обозначаются одинаково:

```
Безусловный переход (jump): JMP op
```

Здесь операнд тем или иным способом указывает адрес перехода, т. е. адрес команды, которая должна быть выполнена следующей. Рассмотрим, что это за способы.

4.1.1. Прямой переход

В данном случае в качестве op указывается метка той команды, на которую надо передать управление:

```
JMP <метка>
```

Пример:

```
   JMP L   ;следующей будет выполняться команда с меткой L
   ...
L: MOV AX,0
```

При программировании на ЯА не надо, как на машинном языке, следить за адресами ячеек, в которые попадают команды, чтобы эти адреса указывать в командах перехода. Достаточно лишь пометить нужную команду и в команде перехода указать ее метку. Адрес же вместо метки подставит сам ассемблер.

Вообще говоря, на этом можно было бы и закончить рассказ про прямой переход, однако для полноты картины следует раскрыть подноготную этой, казалось бы, простой команды.

Напомним, что в ПК имеется регистр IP (указатель команд), в котором всегда хранится адрес той команды, что должна выполняться следующей. Поэтому сделать переход по адресу – значит записать данный адрес в регистр IP. Казалось бы, в команде перехода должен задаваться именно адрес перехода. Однако в ПК машинная команда прямого перехода устроена так, что в ней указывается не этот адрес, а разность между ним и адресом команды перехода. Другими словами, отсчет адреса перехода ведется от команды перехода, в связи с чем такой переход называют относительным. Действие же самой команды перехода заключается в прибавлении этой величины к текущему значению регистра IP.

Замечание. Если говорить точнее, то относительный адрес перехода отсчитывается не от самой команды перехода, а от следующей за ней команды. Дело в том, что в ПК выполнение любой команды начинается с засылки в регистр IP адрес следующей по порядку команды и только затем выполняется собственно команда. Поэтому в команде перехода относительный адрес будет прибавляться к значению IP, которое уже указывает на следующую команду, а потому от этой следующей команды и приходится отсчитывать относительный адрес перехода. Однако в дальнейшем мы не будем обращать внимание на эту деталь, поскольку для ЯА она в общем-то не существенна.

В ПК имеются две машинные команды прямого перехода, в одной из которых относительный адрес перехода задается в виде байта (такая команда называется коротким переходом), а в другой – в виде слова (это команда длинного перехода). В каждой из этих команд операнд рассматривается как целое со знаком (от -128 до +127 или от -2^{15} до $2^{15}-1$), поэтому при сложении его с IP значение этого регистра может как увеличиться, так и уменьшиться, т. е. возможен и переход вперед, и переход назад.

Естественно, возникает вопрос: в чем выгода от того, что указывается не сам адрес перехода, а его расстояние от команды перехода? Это объясняется стремлением создателей ПК сэкономить память, занимаемую командами перехода. Если в команде указывать сам адрес перехода, то на него придется всегда отводить слово (2 байта). Но практика показывает, что в большинстве случаев переходы делаются на команды, расположенные недалеко от команд перехода, поэтому разность между адресами этих двух команд, как правило, небольшая, для нее достаточно и байта. Учитывая это, создатели ПК сделали так, чтобы именно эта разность и указывалась в командах перехода, чтобы большинство команд перехода можно было сделать на один байт короче.

Однако эта экономия оборачивается бедой для программистов: при записи программы на машинном языке приходится вычислять относительные адреса переходов, что является крайне неприятным занятием. Но, к счастью, ЯА избавляет нас от этого занятия: программируя на ЯА, мы указываем лишь метку нужной команды, а уж ассемблер сам подсчитывает разность между

адресом этой метки и адресом команды перехода и подставляет эту разность в машинную команду. Более того, в ЯА в команде JMP вообще нельзя указывать непосредственный операнд, а можно указывать только метку. Так что, программируя на ЯА, можно считать, что в команде перехода указывается сам адрес перехода.

4.1.2. Оператор SHORT

Зачем же мы тогда обо всем этом рассказали? Дело в следующем. Встречая символьную команду перехода с меткой, ассемблер вычисляет разность между адресом этой метки и адресом самой команды перехода и оценивает величину этой разности. Если она небольшая, укладывается в байт, тогда ассемблер формирует машинную команду короткого перехода (она занимает 2 байта), а если разность большая – формирует команду длинного перехода (3 байта). Однако сделать такой выбор ассемблер может, только если метка была описана до команды перехода, т. е. если эта метка является ссылкой назад. Поскольку ассемблер просматривает текст программы сверху вниз, то, дойдя до команды перехода, он будет знать как адрес метки, так и адрес команды перехода и потому сможет вычислить разность между этими адресами, сможет оценить ее величину. Но если в команде перехода указана метка "вперед", то, встретив команду перехода, ассемблер еще не будет знать адреса метки и потому не сможет оценить величину разности, не сможет определить, какой здесь переход – короткий или близкий. Об этом он узнает позже, когда дойдет до описания метки, а пока он "на всякий случай" формирует команду длинного перехода – так он не ошибется.

Однако такой способ трансляции переходов вперед не всегда выгоден: если метка окажется близко расположенной, то мы потеряем байт на этом. Так вот, если мы заранее знаем, что переход вперед будет коротким, и если нам жалко терять байт на команде перехода, то мы должны предупредить ассемблер о том, что переход будет коротким. Для этого в ЯА введен оператор SHORT (короткий), который ставится в команде перехода перед меткой. В этом случае ассемблер сформирует машинную команду короткого перехода:

```
   JMP L          ; длинный переход (3 байта)
   JMP SHORT L    ; короткий переход (2 байта)
   ...
L: ...
```

Отметим, что если мы указали оператор SHORT, но при этом ошиблись (переход на самом деле оказался длинным), тогда ассемблер зафиксирует ошибку. Так что пользоваться оператором SHORT надо осторожно, только если мы твердо уверены, что переход будет коротким. Обычно этот оператор используют лишь тогда, когда сокращение памяти, занимаемой программой, – это вопрос жизни или смерти.

Оператор SHORT можно ставить и перед меткой, описанной раньше, т. е. при переходе назад, однако в этом случае ассемблер проигнорирует этот оператор, поскольку он и так будет знать "расстояние" перехода.

4.1.3. Косвенный переход

Теперь рассмотрим другую разновидность безусловного перехода – косвенный переход. В этом случае в команде перехода указывается не сам адрес перехода, а то место, где находится этот адрес. Таким местом может быть регистр общего назначения или слово памяти:

```
JMP r16    или    JMP m16
```

В этих командах берется содержимое указанного регистра или слова памяти, оно рассматривается как адрес некоторой команды программы и именно по этому адресу делается переход. Причем этот адрес рассматривается как "настоящий", а не отсчитанный от команды перехода.

Примеры ([x] – содержимое ячейки или регистра x):

```
A DW L
  ...
  JMP A    ; goto [A] = goto L
  ...
  MOV DX,A ; DX=L
  JMP DX   ; goto [DX] = goto L
  ...
L: ...
```

Косвенные переходы используются в тех случаях, когда адрес перехода становится известным только во время счета программы. Примеры на косвенные переходы будут приведены позже.

А сейчас рассмотрим одну проблему, с которой сталкивается ассемблер при трансляции команд безусловного перехода. Возьмем команду JMP Z, где Z – некоторое имя (но не имя регистра). Что это такое – прямой переход по метке Z или косвенный переход по адресу из ячейки Z? Если имя Z описано до этой команды, то здесь проблемы нет: если именем Z помечена команда (рис. а), то это переход по метке, а если имя Z описано в директиве DW (рис. б), то это косвенный переход.

```
Z: INC AX            Z DW L              JMP Z ; goto ?
   ...                  ...                ...
   JMP Z ;goto Z        JMP Z ;goto L     Z ...
   а)                   б)                 в)
```

Но если Z – ссылка вперед, т. е. это имя описывается позже (рис. в), тогда ассемблер не будет знать, какой здесь переход. Чтобы снять эту неоднозначность, в ЯА принято следующее соглашение: в подобной ситуации ассемблер всегда считает, что Z – метка, и потому всегда формирует команду прямого перехода по этой метке (причем команду длинного перехода). Если же затем обнаружится, что Z – не метка, то будет зафиксирована ошибка.

Так вот, если нас это правило не устраивает, если нам нужен косвенный переход, то мы обязаны сообщить об этом ассемблеру. Для этого используется уже известный нам оператор PTR: вместо просто имени Z надо записать

конструкцию WORD PTR Z, которой мы сообщаем ассемблеру, чтобы он рассматривал Z как имя переменной размером в слово, чтобы он формировал машинную команду косвенного перехода.

Итак, при переходах вперед имеем следующие случаи:

```
 JMP Z ;goto Z      JMP Z ;ошибка      JMP WORD PRT Z ;goto L
  ...                ...                ...
Z: ...             Z DW L             Z DW L
```

4.2. Команды сравнения и условного перехода

Если переход осуществляется только при выполнении некоторого условия и не осуществляется в противном случае, то такой переход называется условным. Условный переход обычно реализуется в два шага: сначала сравниваются некоторые величины, в результате чего соответствующим образом формируются флаги (ZF, SF и т. д.), а затем выполняется собственно условный переход в зависимости от значений флагов. Поэтому мы сейчас рассмотрим и команду сравнения, и команды условного перехода.

```
Сравнение (compare): CMP op1,op2
```

Эта команда эквивалентна команде SUB op1,op2 за одним исключением: вычисленная разность op1-op2 никуда не записывается. Поэтому единственный и главный эффект от команды сравнения – это установка флагов, характеризующих полученную разность, или, что то же самое, характеризующих сравниваемые величины op1 и op2. Как формируются флаги при вычитании, мы уже рассматривали (см. разд. 3.3), поэтому повторяться не будем.

Что же касается команд условного перехода, то их в ПК достаточно много, но в ЯА они все записываются единообразно:

```
Jxx <метка>
```

где операнд указывает метку той команды программы, на которую надо сделать переход в случае выполнения некоторого условия, а мнемокод начинается буквой J (от jump), за которой следует одна или несколько букв, в сокращенном виде описывающих это условие.

Все команды условного перехода можно разделить на три группы.

В первую группу входят команды, которые ставятся после команды сравнения. В их мнемокодах с помощью определенных букв описывается тот исход сравнения, при котором надо делать переход. Это такие буквы:

E – equal (равно)
N – not (не, отрицание)
G – greater (больше) – для чисел со знаком
L – less (меньше) – для чисел со знаком
A – above (выше, больше) – для чисел без знака
B – below (ниже, меньше) – для чисел без знака

Как видно, для условий "меньше" и "больше" введены две системы обозначений. Это связано с тем, что после сравнения чисел со знаком и сравнения чисел без знака надо реагировать на разные значения флагов.

Отметим, что одна и та же команда условного перехода может иметь в ЯА несколько названий-синонимов. Это объясняется тем, что одно и то же условие перехода может быть сформулировано по-разному. Например, условие "меньше" – это в то же время и условие "не верно, что больше или равно", поэтому переход по меньше для знаковых чисел обозначается и как JL, и как JNGE. Какое из этих названий-синонимов использовать – это личное дело автора программы.

Теперь приведем названия всех команд условного перехода, используемых после команды сравнения (через косую черту указаны названия-синонимы).

Мнемокод	*Содержательное условие для перехода после CMP op1,op2*	*Состояние флагов для перехода*
	Для любых чисел:	
JE	op1=op2	ZF=1
JNE	op1<>op2	ZF=0
	Для чисел со знаком:	
JL/JNGE	op1<op2	SF<>OF
JLE/JNG	op1<=op2	SF<>OF или ZF=1
JG/JNLE	op1>op2	SF=OF и ZF=0
JGE/JNL	op1>=op2	SF=OF
	Для чисел без знака:	
JB/JNAE	op1<op2	CF=1
JBE/JNA	op1<=op2	CF=1 или ZF=1
JA/JNBE	op1>op2	CF=0 и ZF=0
JAE/JNB	op1>=op2	CF=0

(Объясним, к примеру, почему в команде условного перехода "по меньше" для знаковых чисел (JL) проверяется соотношение OF<>SF. Если в команде CMP op1,op2 сравниваемые числа трактуются как знаковые, тогда возможны две комбинации флагов, соответствующие условию op1<op2. Во-первых, если при вычитании op1-op2 не было переполнения мантиссы (OF=0), тогда флаг SF фиксирует настоящий знак разности op1-op2 и потому SF=1 означает, что op1-op2<0, т. е. op1<op2. Во-вторых, если при вычитании произошло переполнение мантиссы (OF=1), тогда результатом команды будет число с протиивоположным знаком, чем у настоящей разности, и поскольку флаг SF фиксирует не знак настоящей разности, а знак результата команды, то условие SF=0 означает, что у искаженного результата знак положителен, а значит, у настоящей разности знак отрицателен, т. е. op1<op2. Итак, условию op1<op2 соответствует либо OF=0 и SF=1, либо OF=1 и SF=0, что можно записать более коротко: OF<>SF. Именно это условие и указано в таблице для команды JL.)

Пример. Пусть X, Y и Z – переменные размером в слово. Требуется записать в Z максимальное из чисел X и Y.

Решение этой задачи различно для чисел со знаком (см. слева) и для чисел без знака (см. справа), т. к. приходится использовать разные команды условного перехода:

```
;числа со знаком                ;числа без знака
  MOV AX,X                      MOV AX,X
  CMP AX,Y     ;x=y?            CMP AX,Y
  JGE M        ;x>=y -> M       JAE M
  MOV AX,Y                      MOV AX,Y
M: MOV Z,AX                   M: MOV Z,AX
```

Во вторую группу команд условного перехода входят те, которые ставятся после команд, отличных от команды сравнения, и которые реагируют на то или иное значение какого-нибудь определенного флага. В мнемокодах этих команд указывается первая буква проверяемого флага, если переход должен быть выполнен при значении 1 у флага, либо эта буква указывается с буквой N (not), если переход надо сделать при нулевом значении флага:

Мнемокод	*Условие перехода*	*Мнемокод*	*Условие перехода*
JZ	ZF=1	JNZ	ZF=0
JS	SF=1	JNS	SF=0
JC	CF=1	JNC	CF=0
JO	OF=1	JNO	OF=0
JP	PF=1	JNP	PF=0

Замечание. Легко заметить, что следующие пары мнемокодов эквиваленты: JE и JZ, JNE и JNZ, JB и JC, JNB и JNC.)

Пример. Пусть A, B и C – беззнаковые байтовые переменные. Требуется вычислить C=A*A+B, но если ответ превосходит размер байта, тогда надо передать управление на метку ERROR.

Возможное решение этой задачи:

```
MOV AL,A
MUL AL
JC ERROR   ;A*A > 255 (CF=1) --> ERROR
ADD AL,B
JC ERROR   ;перенос (CF=1) --> ERROR
MOV C,AL
```

И, наконец, в третью группу входит только одна команда условного перехода, проверяющая не флаги, а значение регистра CX:

```
JCXZ <метка>
```

Действие команды JCXZ (jump if CX is zero) можно описать так:

```
if CX=0 then goto <метка>
```

Примеры на использование этой команды будут приведены позже.

Отметим общую особенность команд условного перехода: все они осуществляют только короткий переход, т. е. с их помощью можно передать управление не далее чем на 127–128 байт вперед или назад. Это примерно

30–40 команд (в среднем одна команда ПК занимает 3–4 байта). Дело в том, что в ПК все машинные команды условного перехода имеют вид КОП i8 и реализуют короткий относительный переход: IP:=IP+i8, а команд с операндом в слово (i16) нет. Это объясняется тем, что в большинстве случаев как раз и нужны такие короткие переходы, а с другой стороны, для команд длинных условных переходов в ПК попросту не хватило кодов операций.

Естественно, возникает вопрос: а как в этих условиях осуществлять длинные условные переходы, на расстояние более 127 байт от команды перехода? Здесь надо привлекать команду длинного безусловного перехода. Например, при "далекой" метке M оператор

```
if AX=BX then goto M
```

следует реализовывать так:

```
  if AX<>BX then goto L;   {короткий переход}
  goto M;                  {длинный переход}
L: ...
```

На ЯА это записывается следующим образом:

```
  CMP AX,BX
  JNE L
  JMP M
L: ...
```

Получается очень коряво, то иного варианта нет.

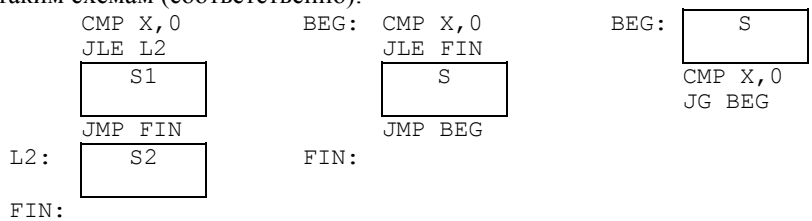
Отметим, что использовать в командах условного перехода оператор SHORT не надо, т. к. все эти переходы и так короткие.

4.3. Команды управления циклом

С помощью команд перехода можно реализовать любые разветвления и циклы. Например, следующие операторы языка Паскаль

а) if X>0 then S1 else S2
б) while X>0 do S
в) repeat S until X>0

где S, S1 и S2 – какие-то операторы, а X – знаковая переменная, реализуются по таким схемам (соответственно):



а) б) в)

Среди циклов на практике наиболее часто встречаются циклы с заранее известным числом повторений (for-циклы языка Паскаль), поэтому в систему команд многих ЭВМ обычно включают дополнительные команды, упрощающие реализацию подобных циклов и называемые командами управления циклом. Есть такие команды и в ПК, их мы сейчас и рассмотрим.

4.3.1. Команда LOOP

Пусть некоторую группу команд (тело цикла) надо повторить N раз (N>0). Тогда на ЯА этот цикл можно реализовать по такой схеме:

```
   MOV CX,N  ;CX – счетчик цикла (число повторений)
L: ...       ;
   ...       ; тело цикла
   ...       ;
   DEC CX    ;CX:=CX-1
   CMP CX,0  ;CX=0?
   JNE L     ;CX<>0 -> goto L
```

Как видно, в конце таких циклов всегда применяется одна и та же тройка команд. Учитывая это, в систему команд ПК была введена специальная команда, которая объединяет в себе действия этих трех команд:

```
Управление циклом по счетчику: LOOP <метка>
```

Действие этой команды можно описать так:

```
CX:=CX-1; if CX<>0 then goto <метка>
```

С помощью команды LOOP наш цикл запишется следующим образом:

```
   MOV CX,N  ;N>0
L: ...       ;
   ...       ; тело цикла
   ...       ;
   LOOP L
```

Как видно, получилось короче, да и работает команда LOOP быстрее этих трех команд. Поэтому, если можно, следует пользоваться командой LOOP.

Однако необходимо учитывать ряд особенностей этой команды.

Во-первых, команда LOOP требует, чтобы в качестве счетчика цикла обязательно использовался регистр CX, при другом регистре команду применять нельзя.

Во-вторых, начальное значение для CX должно быть присвоено до цикла, причем этому регистру надо присваивать ровно столько, сколько раз должен повторяться цикл – без всяких "плюс-минус единица". Например, если цикл должен выполняться 100 раз, то в регистр CX надо записывать именно 100, а не 99 или 101.

В-третьих, поскольку команда LOOP ставится в конце цикла, тело цикла хотя бы раз обязательно выполнится. Поэтому для случая CX=0 наша схема цикла не подходит. Так вот, если возможен вариант, что число повторений может быть и нулевым, то при CX=0 надо сделать обход цикла:

```
   MOV CX,N  ; N>=0
   JCXZ L1   ; CX=0 -> L1
L: ...       ;
   ...       ; тело цикла
   ...       ;
   LOOP L
L1: ...
```

Именно ради осуществления таких обходов в ПК и была введена команда условного перехода JCXZ. В иных ситуациях она используется редко.

В-четвертых, как и команды условного перехода, команда LOOP реализует только короткий переход, поэтому расстояние от нее до начала цикла (метки L) не должно превышать 127–128 байт (примерно 30–40 команд). Если цикл содержит больше команд, тогда команду LOOP использовать нельзя и надо реализовывать цикл по-иному.

Рассмотрим конкретный пример на использование команды LOOP. Пусть N – байтовая переменная со значением от 0 до 8 и надо в регистр AX записать факториал этого числа: AX:=N! (отметим, что $8! = 40320 < 2^{16}$).

Для решения этой задачи надо вначале положить AX:=1, а затем N раз выполнить умножение AX:=AX*i, меняя i от 1 до 8. При этом следует учитывать, что при N=0 цикл не должен выполняться.

```
   MOV AX,1  ;AX:=0!
   MOV CL,N
   MOV CH,0  ;CX:=N как слово (счетчик цикла)
   JCXZ F1   ;при N=0 обойти цикл
   MOV SI,1  ;i:=1
F: MUL SI    ;(DX,AX):=AX*i (DX=0)
   INC SI    ;i:=i+1
   LOOP F
F1:
```

Отметим, что в данном примере параметр цикла (i) можно было бы менять и в "обратном" направлении – от N до 1, а поскольку именно так меняется и регистр CX, то этот регистр можно использовать не только как счетчик цикла, но и как параметр цикла (CX=i):

```
   MOV AX,1
   MOV CL,N
   MOV CH,0   ;CX – и счетчик цикла, и параметр цикла
   JCXZ F1
F: MUL CX
   LOOP F
F1:
```

Однако такое совмещение ролей удается крайне редко, поэтому при реализации циклов регистр CX обычно используется только как счетчик цикла (указывающий, сколько еще раз надо повторить цикл), а для параметра цикла приходится использовать иной регистр.

4.3.2. Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ

Эти команды похожи на команду LOOP, т. е. заставляют цикл повториться столько раз, сколько указано в регистре CX, однако они допускают и досрочный выход из цикла.

```
Цикл по счетчику и пока равно (пока ноль):LOOPE <метка> или
LOOPZ <метка>. (Названия LOOPE и LOOPZ являются синонимами.)
```

Действие этой команды можно описать так:

```
CX:=CX-1; If (CX<>0) and (ZF=1) then goto <метка>
```

Таким образом, эта команда совмещает в себе изменение счетчика цикла (регистра CX) и условный переход, когда счетчик еще не нулевой и когда предыдущая команда выработала флаг нуля, равный 1. Причем этот переход – короткий.

Команда LOOPE используется для организации цикла с известным числом повторений, из которого возможен досрочный выход. До начала цикла в регистр CX записывается число повторений. Сама команда LOOPE ставится в конец цикла (ее операнд – метка первой команды цикла), а перед ней помещается команда, меняющая флаг ZF (обычно это команда сравнения CMP). Команда LOOPE заставляет цикл повторяться CX раз, но только если предыдущая команда фиксирует равенство сравниваемых величин (вырабатывает нулевой результат).

По какой именно причине произошел выход из цикла (по ZF=0 или CX=0), надо проверять после цикла. Причем надо проверять флаг ZF (по команде JE/JZ или JNE/JNZ), а не регистр CX, т. к. условие ZF=0 ("не равно") может появиться как раз на последнем шаге цикла, когда и регистр CX стал нулевым.

Чаще всего команда LOOPE используется для поиска первого элемента некоторой последовательности, отличного от заданной величины. Пусть, к примеру, надо записать в регистр BL наименьшее число из отрезка [2, K], на которое не делится число N (K и N – байтовые переменные, 2<=K<N), или записать 0, если такого числа нет. Для этого будем последовательно делить N на числа 2, 3, ..., K и сравнивать остатки от деления с 0 – до тех пор, пока не найдется ненулевой остаток либо не будут исчерпаны все числа отрезка:

```
    MOV DL,N
    MOV DH,0   ;DX:=N как слово
    MOV CL,K
    MOV CH,0
    DEC CX     ;CX:=K-1 (счетчик цикла)
    MOV BL,1
DV: INC BL     ;очередное число из [2,K]
    MOV AX,DX
    DIV BL     ;AH:=N mod BL
    CMP AH,0   ;mod=0?
    LOOPE DV   ;цикл CX раз и пока mod=0
    JNE DV1    ;mod<>0 --> DV1
    MOV BL,0   ;нет искомого числа
DV1:
```

Еще одна команда ПК для организации циклов:

```
Цикл по счетчику и пока не равно (пока не ноль): LOOPNE <метка> или LOOPNZ <метка>. (Названия LOOPNE и LOOPNZ являются синонимами.)
```

Эта команда аналогична команде LOOPE/LOOPZ, но выход из цикла осуществляет при CX=0 или ZF=1 (если предыдущая команда зафиксировала равенство, дала нулевой результат). Ее действие:

```
CX:=CX-1; if (CX<>0) and (ZF=0) then goto <метка>
```

Команда LOOPNE обычно используется для поиска в некоторой последовательности первого элемента, имеющего заданную величину.

4.4. Вспомогательные операции ввода-вывода

Для того чтобы написать пусть и простую, но законченную программу, необходимо знать, по крайней мере, три вещи – как вводить исходные данные, как выводить результаты и как остановить выполнение программы.

Позже (в гл. 13) мы узнаем, что в ПК с помощью имеющихся команд можно ввести и вывести только один символ, а, например, ввести и вывести число уже нельзя. Однако, чтобы можно было составлять содержательные программы, нам уже сейчас нужны достаточно мощные операции ввода-вывода. Поэтому предлагается следующее: в гл. 13 будут определены некоторые операции ввода-вывода и операция останова программы, но мы уже сейчас будем ими пользоваться. Обращения к этим операциям выглядят как обычные команды, поэтому можно считать, что в ПК как будто бы есть такие команды. Они и будут рассмотрены в этом разделе.

Предварительно лишь отметим одну особенность этих команд: если запись их операнда содержит пробел, тогда этот операнд обязательно должен быть заключен в угловые скобки. Например, вместо

```
OUTINT WORD PTR X,7
```

надо писать:

```
OUTINT <WORD PTR X>,7
```

Причина этого станет понятной позже, когда в гл. 11 мы познакомимся с макрокомандами (команды ввода-вывода – это на самом деле макрокоманды).

4.4.1. Останов программы

Когда программа выполнит все необходимые действия, то ее надо остановить. Для этого используется команда

```
FINISH
```

Эта команда обязательно должна быть выполнена, иначе программа не остановится.

4.4.2. Ввод с клавиатуры

Ввод данных с клавиатуры реализован с использованием промежуточного буфера ввода (специальной области памяти): все набираемые на клавиатуре символы сначала попадают в этот буфер (после нажатия на клавиатуре клавиши Enter), и уже отсюда они затем будут считываться командами ввода. Это означает, что можно досрочно ввести много данных и они не пропадут. Например, если в программе осуществляется ввод по одному символу, то все равно можно сразу набрать много символов – затем они будут считываться по одному. Кроме того, при вводе (пока не нажата клавиша Enter) допускается редактирование вводимого текста: при нажатии клавиши "стрелка влево" или Backspace уничтожается последний набранный символ, а при нажатии клавиши Esc уничтожается весь набранный текст.

Отметим также, что команды ввода не выдают на экран никакого приглашения к вводу, поэтому сама программа должна выдавать символы приглашения (например, символ >).

Теперь перечислим команды ввода.

```
Ввод символа: INCH op
```

Допустимые типы операнда: r8, m8.

По этой команде вводится очередной символ и он (точнее, его код) записывается в указанный байтовый регистр или байт памяти.

```
Ввод числа: ININT op
```

Допустимые типы операнда: r16, m16.

По команде можно ввести число как со знаком, так и без знака. Вводимое число должно быть записано в десятичной системе. Если перед числом имеются пробелы, то они "проглатываются" командой ININT. Если число набрано без знака или со знаком "плюс", то оно вводится как число без знака и может иметь величину от 0 до 2^{16}-1. Если же перед числом указан знак "минус", то оно вводится как отрицательное число (записывается в дополнительном коде) и должно иметь величину от -2^{15} до -1. Концом числа считается любой символ, отличный от цифры. Если величина числа находится вне указанных границ или оно задано неправильно, то фиксируется ошибка и программа прекращает свою работу.

Отметим, что введенное число всегда записывается в регистр или ячейку размером в слово, но не байт. Поэтому, например, команда ININT AX допустима, а команда ININT AH – нет.

```
Очистка буфера ввода: FLUSH
```

Это аналог процедуры readln языка Паскаль: все, что до этого момента было набрано на клавиатуре, уничтожается.

4.4.3. Вывод на экран

Вывод на экран осуществляется немедленно, без каких-либо промежуточных буферов. Очередной выводимый символ размещается в той позиции экрана, где сейчас находится курсор, который по мере вывода смещается.

```
Переход на новую строку: NEWLINE
```

Аналог процедуры writeln языка Паскаль: по этой команде курсор перемещается на начало следующей строки экрана.

```
Вывод символа: OUTCH op
```

Допустимые типы операнда: i8, r8, m8.

По этой команде на экране высвечивается указанный символ. Символ задается либо в самой команде, либо в байтовом регистре, либо в ячейке памяти размером в байт.

```
Вывод строки: OUTSTR
```

Как видно, в этой команде не указывается операнд, но он есть, его местонахождение фиксировано: в регистре DX должен находиться начальный адрес выводимой строки. Кроме того, сама строка должна оканчиваться символом '$', по которому и определяется ее конец. По OUTSTR выводятся все символы строки, начиная с указанного адреса и до первого знака $, который уже не выводится.

Пример:

```
 S DB 'Hello','$'
AS DW S          ;начальный адрес строки S
   ...
   MOV DX,AS     ;DX:=адрес S
   OUTSTR        ;вывод: Hello
```

(Позже мы увидим, как можно более простым способом заслать адрес строки в регистр.)

```
Вывод числа со знаком: OUTINT op1 [,op2]
Вывод числа без знака: OUTWORD op1 [,op2]
```

Допустимые типы операндов: op1: i16, r16, m16; op2: i8, r8, m8.

Обе эти команды действуют аналогично – выводят в десятичном виде число, заданное первым операндом и имеющее размер слова, только команда OUTINT трактует его как знаковое число, а команда OUTWORD – как беззнаковое. Например:

```
OUTINT 0FFFFh   ; вывод -1
OUTWORD 0FFFFh  ; вывод 65535
```

Второй операнд, если есть, всегда трактуется как число без знака и задает ширину поля вывода – число позиций на экране, которые отводятся для вывода числа. Если ширина поля вывода больше, чем надо, то число прижимается к правому краю этого поля, а перед числом ставятся пробелы. Если же ширина поля меньше или вообще не указана (нет op2), тогда никакие пробелы не выводятся, а выводится только число, причем целиком, без каких-либо

усечений. Другими словами, эти команды аналогичны оператору write(op1:op2) языка Паскаль.

Например, по командам:

```
OUTCH '*'
OUTINT 12
OUTWORD 345,6
OUTINT -67
```

будет выдана следующая строка (для наглядности пробелы заменены знаками подчеркивания):

```
* 1 2 _ _ _ 3 4 5 - 6 7
```

4.5. Примеры

В разделе приведены примеры использования рассмотренных в этой главе команд перехода и управления циклами, а также операций ввода-вывода и указаны их решения с необходимыми пояснениями.

Пример 1

Если в регистре BL находится код шестнадцатеричной цифры (от '0' до '9' или от 'A' до 'F'), тогда требуется записать в BL числовую величину этой цифры (от 0 до 15).

Решение

В системах кодировки, используемых в ПК, для любой цифры d от 0 до 9 справедливо соотношение код(d)-код('0')=d, а для любой большой латинской буквы l разность код(l)-код('A') равна номеру этой буквы (при нумерации с 0) в латинском алфавите. Поэтому в нашей задаче значение регистра BL надо уменьшить на код символа '0', если в BL находится код обычной цифры (от '0' до '9'), или уменьшить на код буквы 'A' и затем увеличить на 10 ('A' → 10, 'B' → 11, ...), если в BL находится код "буквенной" шестнадцатеричной буквы.

Напомним, что коды символов являются беззнаковыми числами, поэтому для сравнения символов на "больше" или "меньше" надо воспользоваться командами JA или JB.

```
     CMP BL,'0'      ;проверка на обычную цифру
     JB LET
     CMP BL,'9'
     JA LET
     SUB BL,'0'      ;'0'<=BL<='9' --> BL:=BL-код('0')
     JMP FIN
LET: CMP BL,'A'      ;проверка на "буквенную" цифру
     JB FIN
     CMP BL,'F'
     JA FIN
     ADD BL,-'A'+10  ;'A'<=BL<='F' --> BL:=BL-код('A')+10
FIN:
```

Пример 2

A, B и C – переменные размером в слово, причем A>0 и B>0. Требуется записать в C наибольший общий делитель чисел A и B: C=НОД(A,B).

Решение

Для нахождения НОД двух чисел воспользуемся алгоритмом Евклида в следующем варианте: пока эти числа не равны, надо вычитать из большего числа меньшее, а когда они сравняются, то это и будет НОД исходных чисел. (Этот алгоритм можно применять только к строго положительным числам, иначе он зациклится.)

Кроме того, воспользуемся тем обстоятельством, что команды перехода не меняют флаги, поэтому после команды сравнения можно выполнять любое число команд условного перехода для проверки флагов, установленных этой одной командой сравнения.

```
      MOV AX,A      ;запись копий чисел в регистры AX и BX
      MOV BX,B
NOD:  CMP AX,BX     ;AX=BX?
      JE NOD2       ;AX=BX --> выход из цикла (флаги не изменились)
      JA NOD1       ;AX>BX --> NOD1
      XCHG AX,BX    ;записать в AX большее число , а в BX меньшее
NOD1: SUB AX,BX     ;из большего числа вычесть меньшее
      JMP NOD       ;в цикл
NOD2: MOV C,AX      ;C:=НОД(A,B)
```

Пример 3

K – байтовая переменная со значением от 1 до 18. Требуется записать в регистр AL количество двузначных десятичных чисел (от 10 до 99), сумма цифр которых равна K.

Решение

Можно было бы, конечно, перебрать все числа от 10 до 99 и для каждого из них с помощью команды деления выделить обе его цифры. Однако, чтобы не связываться с командой деления, лучше организовать вложенный цикл: внешний – по левой цифре, а внутренний – по правой цифре двузначных чисел. На языке Паскаль этот цикл выглядит так:

```
for dh:=1 to 9 do
   for dl:=0 to 9 do
      if dh+dl=k then al:=al+1
```

Для организации обоих циклов мы будем использовать команду LOOP. Но это значит, что каждый из циклов должен использовать регистр CX как свой счетчик цикла, поэтому циклы будут "мешать" друг другу, будут портить значение CX другого цикла. Устранить эту неприятность можно следующим образом: перед началом выполнения внутреннего цикла надо спасти где-то (например, в другом регистре) значение CX, соответствующее внешнему циклу, и затем использовать CX для организации внутреннего цикла, а по его окончанию надо восстановить в CX прежнее значение.

С учетом всего сказанного наша задача решается так:

```
    MOV AL,0    ;количество искомых чисел
    MOV CX,9    ;счетчик внешнего цикла
    MOV DH,1    ;левая цифра
L0: MOV BX,CX   ;спасти CX
  ; начало внутреннего цикла
    MOV CX,10   ;счетчик внутреннего цикла
    MOV DL,0    ;правая цифра
L1: MOV AH,DH
    ADD AH,DL
    CMP AH,K    ;левая цифра + правая цифра = K ?
    JNE L2
    INC AL      ;учесть число с суммой цифр, равной K
L2: INC DL      ;увеличить правую цифру
    LOOP L1
  ; конец внутреннего цикла
    MOV CX,BX   ;восстановить CX для внешнего цикла
    INC DH      ;увеличить левую цифру
    LOOP L0
```

Пример 4

Для ввода задана последовательность из 200 знаковых чисел (размером в слово). Требуется определить, сколько раз наименьшее из этих чисел встречается в данной последовательности, и вывести в одной строке экрана ПК само минимальное число и количество его вхождений.

Решение

Введем первое число и будем пока считать его минимальным и встретившимся только раз. Затем в цикле вводим остальные 199 чисел и для каждого из них выполняем следующие действия: если оно больше минимального, то ничего не делаем, если равно минимальному, то увеличиваем количество его вхождений на 1, а если оно меньше минимального, тогда запоминаем это число как новый минимум с количеством вхождений 1.

По завершению цикла минимальное число выводим как знаковое (по OUTINT), а число его вхождений как беззнаковое (по OUTWORD), причем, чтобы эти два числа не "слиплись", для второго числа указываем ширину поля вывода с запасом.

```
      N EQU 200     ;количество вводимых чисел
        OUTCH '>'   ;приглашение к вводу чисел
        ININT AX    ;как min берется первое из чисел
        MOV BX,1    ;количество его вхождений
        MOV CX,N-1 ;счетчик цикла (количество оставшихся чисел)
    IN: ININT DX    ;ввод очередного числа
        CMP DX,AX
        JG NEXT     ;очередное число > min --> NEXT
        JL NEWMIN   ;очередное число < min --> NEWMIN
        INC BX      ;учесть повторное вхождение min
        JMP NEXT
NEWMIN: MOV AX,DX   ;сменить min
```

```
        MOV BX,1
  NEXT: LOOP IN         ;цикл N-1 раз
        OUTINT AX       ;вывод min (как знакового)
        OUTWORD BX,5    ;вывод числа вхождений (как беззнакового)
        NEWLINE         ;перевод строки
        FINISH
```

Пример 5

Ввести последовательность (возможно, пустую) символов, за которой следует точка (это признак конца последовательности), и определить, упорядочена ли эта последовательность по неубыванию кодов символов. В качестве ответа выдать "УПОРЯДОЧЕНО" или "НЕ УПОРЯДОЧЕНО".

Решение

Будем вводить по одному символу (до точки), храня при этом предыдущий символ последовательности. Если эта пара символов неправильно упорядочена, тогда прекращаем цикл, очищаем буфер ввода от символов, которые пользователь мог набрать загодя, и выдаем ответ. Если же пара соседних символов упорядочена как надо, тогда очередной символ запоминаем как предыдущий и повторяем цикл. Если в качестве очередного символа была введена точка, то это значит, что последовательность упорядочена.

Ответ будем выводить как строку (по OUTSTR). Для этого описываем в программе строку "НЕ УПОРЯДОЧЕНО", за которой следует знак $, и записываем в регистр DX начальный адрес этой строки, для чего определяем в программе еще одну переменную, начальным значением которой является адрес этой строки. Если исходная последовательность не упорядочена, тогда сразу выполняем операцию OUTSTR, а иначе увеличиваем значение регистра DX на 3, чтобы пропустить первые три символа нашей строки, чтобы DX указывал на начало подстроки "УПОРЯДОЧЕНО", и только после этого выполняем OUTSTR.

```
 ANS DB "НЕ УПОРЯДОЧЕНО","$"
  AA DW ANS       ;адрес строки ANS
     ...
     MOV DX,AA    ;DX – на начало строки ABS
     OUTCH '>'    ;приглашение к вводу
     MOV AL,0     ;предыдущий символ (вначале – с кодом 0)
 IN: INCH AH      ;очередной символ
     CMP AH,'.'
     JE YES       ;введена точка --> YES
     CMP AL,AH    ;сравнить предыдущий и очередной символы
     JA NO        ;неправильный порядок --> NO
     MOV AL,AH    ;очередной символ становится предыдущим
     JMP IN       ;в цикл
 NO: FLUSH        ;очистка буфера ввода от оставшихся символов
     JMP OUT
YES: ADD DX,3     ;DX – на начало подстроки "УПОРЯДОЧЕНО"
OUT: OUTSTR       ;вывод ответа
     FINISH
```

Пример 6

Осуществить посимвольный ввод (использовать только команду INCH) десятичного числа и записать его в переменную N размером в слово.

Предполагается, что число имеет величину от -2^{15} до $2^{16}-1$ и задано для ввода без ошибок: в нем указаны только цифры, причем не менее одной, перед числом возможен знак + или -. Признаком конца числа считать пробел.

Решение

Сначала введем первый символ и проверим, знак ли это. Если это плюс или знака нет, тогда запоминаем, что это число положительно, а при минусе фиксируем отрицательность числа. Знак числа будет учтен в конце, после ввода всех цифр числа. Если число начинается со знака, тогда вводим следующий символ, чтобы в любом случае уже была введена первая цифра числа.

Далее по очереди вводим цифры и формируем число по следующей схеме (схеме Горнера). Пусть уже введены первые цифры числа, скажем 3 и 7, и по ним сформировано число 37 и пусть следующей введена цифра 4. Тогда умножаем предыдущее число на 10 и добавляем к нему новую цифру: 37*10+4=374. И так делаем для каждой новой цифры: умножаем ранее полученное число на 10 и прибавляем эту цифру.

Этот цикл заканчивается, когда будет введен признак конца числа, т. е. пробел. Далее учитываем знак числа: заменяем число на противоположное, если в начале числа был минус.

```
;анализ знака числа
 SGN: MOV SI,1    ;SI=1 – для неотрицательного числа
      INCH AL
      CMP AL,'+'
      JE DIG1
      CMP AL,'-'
      JNE DIGS    ;нет знака -> в AL уже есть первая цифра
      MOV SI,0    ;SI=0 – для отрицательного числа
DIG1: INCH AL     ;после знака ввести первую цифру
;ввод цифр и формирование модуля числа в регистре AX
DIGS: SUB AL,'0'
      MOV AH,0    ;AX:=первая цифра как число размером в слово
      MOV BX,10   ;множитель (основание системы счисления)
      MOV CH,0    ;нужно для CL -> CX (см. ниже)
  D1: INCH CL     ;ввод очередного символа
      CMP CL,' '
      JE D2       ;пробел --> конец ввода
      SUB CL,'0'  ;очередная цифра как число
      MUL BX      ;(DX,AX):=10*AX (DX=0 – см. условие задачи)
      ADD AX,CX   ;AX:=AX+CL (CL=CX)
      JMP D1
;учет знака числа
  D2: CMP SI,1
      JE D3
      NEG AX      ;был минус – замена знака
  D3: MOV N,AX
```

5. МАССИВЫ. СТРУКТУРЫ

В этой главе рассматриваются способы описания и обработки на ЯА таких составных объектов, как массивы и структуры.

5.1. Об индексах элементов массива

Мы уже знаем, что в ЯА массивы описываются по директивам определения данных с использованием конструкции повторения DUP. Однако кое-что здесь требует уточнения.

Пусть имеется массив X из 30 элементов-слов:

```
X DW 30 DUP(?)
```

Как видно, при описании массива указывается количество элементов в нем и их тип, но не указывается, как нумеруются (индексируются) его элементы. Поэтому такому описанию может соответствовать и массив, в котором элементы нумеруются с 0, т. е. X[0..29], и массив, в котором нумерация начинается с 1, т. е. X[1..30], и массив с любым другим начальным индексом k, т. е. X[k..29+k]. Таким образом, автор программы может "накладывать" на массив различные диапазоны изменения индекса. Какой диапазон выбрать? Иногда границы изменения индекса могут жестко определяться условиями задачи (например, в задаче явно сказано, что элементы нумеруются с 1). Но если нумерация не навязывается извне, тогда лучше выбрать нумерацию с 0. Почему?

Чтобы ответить на этот вопрос, рассмотрим, как зависит адрес элемента массива от индекса этого элемента. Пусть мы решили, что элементы массива X нумеруются с k:

```
X DB 30 DUP(?)  ;X[k..29+k]
```

Тогда верно следующее соотношение:

```
адрес(X[i]) = X+2*(i-k)
```

или в более общем виде, где явный размер (2) элементов массива завуалирован:

```
адрес(X[i]) = X+(type X)*(i-k)
```

Эта зависимость становится наиболее простой при k=0:

```
адрес(X[i]) = X+(type X)*i
```

Поэтому обычно и считают при программировании на ЯА, что элементы массива нумеруются с 0:

```
X DW 30 DUP(?)  ;X[0..29]
```

В дальнейшем именно так мы и будем делать.

Для многомерных массивов ситуация аналогична. Пусть, к примеру, имеется двумерный массив (матрица) A, в котором N строк и M столбцов (N и

M – константы) и все элементы – двойные слова, причем строки нумеруются с k1, а столбцы – с k2:

```
A DD N DUP(M DUP(?))    ;A[k1..N+(k1-1), k2..M+(k2-1)]
```

Здесь мы предполагаем, что элементы матрицы размещаются в памяти по строкам: первые M ячеек (двойных слов) занимают элементы первой строки матрицы, следующие M ячеек – элементы второй строки и т. д. Конечно, элементы матрицы можно размещать и по столбцам, но традиционно принято построчное размещение; его мы и будем придерживаться.

При этом предположении зависимость адреса элемента матрицы от индексов элемента выглядит так:

```
адрес(A[i,j]) = A+M*(type A)*(i-k1)+(type A)*(j-k2)
```

И здесь наиболее простой вид эта зависимость приобретает при нумерации с 0, при k1=0 и k2=0:

```
адрес(A[i,j]) = A+M*(type A)*i+(type A)*j
```

5.2. Реализация переменных с индексом

Следующий вопрос, который возникает при работе с массивами, – это как осуществляется доступ к их элементам, как реализуются переменные с индексом. Чтобы ответить на этот вопрос, надо предварительно познакомиться с такой особенностью ЭВМ, как модификация адресов.

5.2.1. Модификация адресов

До сих пор мы рассматривали команды, в которых для операндов из памяти указывались их точные адреса (имена), например: MOV CX,A. Однако в общем случае в команде вместе с адресом может быть указан в квадратных скобках некоторый регистр, например: MOV CX,A[BX]. Тогда команда будет работать не с указанным в ней адресом A, а с так называемым исполнительным (другое название – эффективным) адресом Аисп, который вычисляется по следующей формуле:

$$Аисп = (A + [BX]) \bmod 2^{16},$$

где [BX] обозначает содержимое регистра BX. Другими словами, прежде чем выполнить команду, центральный процессор прибавит к адресу A, указанному в команде, текущее содержимое регистра BX, получит некоторый новый адрес и именно из ячейки с этим адресом возьмет второй операнд. (Сама команда при этом не меняется, суммирование происходит внутри центрального процессора.) Если в результате сложения получилась слишком большая сумма, то от нее берутся только последние 16 бит, на что и указывает операция mod в приведенной формуле. (Отметим, что если в команде рядом с адресом не указан регистр, то исполнительный адрес считается равным адресу из команды.)

Замена адреса из команды на исполнительный адрес называется модификацией адреса, а регистр, участвующий в модификации, принято называть регистром-модификатором или просто модификатором. Отметим при этом, что в ПК в качестве модификатора можно использовать не любой регистр, а только один из следующих четырех: BX, BP, SI или DI.

Возьмем, к примеру, команду ADD A[SI],5. Здесь в роли модификатора выступает регистр SI. Пусть сейчас в нем находится число 100. Тогда Аисп=A+[SI]=A+100. Значит, по данной команде число 5 будет прибавлено к числу из ячейки с адресом A+100, а не из ячейки с адресом A. Если же в SI находится величина -2 (0FFFEh), тогда Аисп=A-2 и потому число 5 будет складываться с числом из ячейки с адресом A-2.

Этот пример показывает, что одна и та же команда может работать с разными адресами. Это очень важно, ради этого и вводится модификация адресов. Одним из случаев, где полезна модификация адресов, является индексирование, используемое для реализации переменных с индексом.

5.2.2. Индексирование

Рассмотрим следующий пример: пусть имеется массив

```
X DW 100 DUP(?)  ;X[0..99]
```

и требуется записать в регистр AX сумму его элементов.

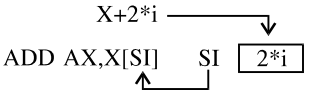
Для нахождения суммы надо сначала в AX записать 0, а затем в цикле выполнять операцию AX:=AX+X[i] при i от 0 до 99. Поскольку адрес элемента X[i] равен X+2*i, то команда, соответствующая этой операции, должна быть следующей:

```
ADD AX,X+2*i
```

Но такая команда запрещена правилами и машинного языка, и ЯА: в любой команде все ее части, в том числе и адрес, должны быть фиксированными, не должны меняться. У нас же адрес меняется вместе с изменением индекса i.

Итак, мы столкнулись со следующей проблемой: по алгоритму наша команда должна работать с разными адресами (как говорят, должна работать с переменным адресом), а правила машинного языка допускают только фиксированный адрес. Именно для устранения этого противоречия и была введена в вычислительные машины модификация адресов, с помощью которой эта проблема решается следующим образом.

Разобьем переменный адрес X+2*i на два слагаемых – на постоянное слагаемое X, которое не зависит от индекса i, и на переменное слагаемое 2*i, зависящее от индекса. Постоянное слагаемое записываем в саму команду, а переменное слагаемое заносим в какой-нибудь регистр-модификатор (скажем, в SI) и название этого регистра также записываем в команду в качестве модификатора:



Что получилось? Поскольку регистр-модификатор один и тот же, то такая команда имеет фиксированный вид, т. е. удовлетворяет правилам машинного языка. С другой стороны, команда работает с исполнительным адресом, а он получается сложением адреса X из команды с содержимым (2*i) регистра SI, которое может меняться. Поэтому, меняя значение SI, мы заставим неменяющуюся команду работать с разными адресами. Тем самым удовлетворяются требования как машинного языка, так и алгоритма. Единственное, что осталось сделать, – это правильно менять содержимое регистра SI. Но это уже делается просто: вначале в SI надо заслать 0, а затем увеличивать его значение с шагом 2; в результате наша команда будет работать с адресами X, X+2, X+4, ..., X+198.

Поскольку в данном случае регистр-модификатор используется для хранения индекса (точнее – выражения, зависящего от индекса), то такой регистр называют индексным регистром, а описанный способ получения адреса переменной с индексом – индексированием.

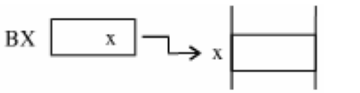
С учетом всего сказанного фрагмент программы нахождения суммы элементов массива X выглядит так:

```
   MOV AX,0       ;начальное значение суммы
   MOV CX,100     ;счетчик цикла
   MOV SI,0       ;начальное значение (удвоенного) индекса
L: ADD AX,X[SI]   ;AX:=AX+X[i]
   ADD SI,2       ;следующий индекс
   LOOP L         ;цикл 100 раз
```

5.2.3. Косвенные ссылки

Рассмотрим еще один случай применения модификации адресов.

Пусть нам надо решить следующую задачу: имеется некоторая ячейка размером в слово, адрес которой нам не известен, но известно, что этот адрес находится в регистре BX, и надо записать, скажем, число 300 в эту ячейку:



Если бы мы заранее знали адрес (x) этой ячейки, то наша задача решалась бы командой MOV x,300. Но в момент составления программы мы не знаем этот адрес, а потому и не можем указать его в команде. Что делать? Вспомним, что команды работают с исполнительными адресами, поэтому нам надо взять такой адрес и такой модификатор, чтобы в сумме они давали этот заранее не известный нам адрес. Легко сообразить, что в нашем случае надо взять нулевой адрес и регистр BX, поскольку тогда Аисп=0+[BX]=0+x=x. Поэтому наша задача решается командой

```
MOV [BX],300
```

Особенность используемого здесь способа модификации адреса заключается в том, что мы, как и прежде, представляем адрес в виде суммы двух слагаемых, одно из которых записываем в команду, а другое – в регистр, но если ранее у нас оба слагаемых были ненулевыми, то теперь одно слагаемое, которое мы помещаем в команду, нулевое и потому весь адрес "упрятан" в регистре. Получается, что в команде мы указывается лишь место (регистр), где находится адрес. Такой способ задания адреса через промежуточное звено называют косвенной ссылкой или косвенной адресацией.

Отметим попутно, что при косвенной ссылке обычно приходится уточнять размер ячейки, на которую она указывает. Если, к примеру, в ячейку, адрес которой находится в регистре BX, надо записать число 0, тогда использовать для этого команду MOV [BX],0 нельзя, т. к. в ней непонятны размеры операндов: 0 может быть как байтом, так и словом, да и адрес из BX также может быть адресом как байта, так и слова (в приведенном выше примере такой неоднозначности не было, т. к. число 300 может быть только словом). Поэтому с помощью оператора PTR надо указать, операнды какого размера мы имеем в виду:

```
MOV BYTE PTR [BX],0 ;пересылка байта
MOV WORD PTR [BX],0 ;пересылка слова
```

5.2.4. Модификация по нескольким регистрам

Мы рассмотрели ситуации, когда модификация адресов осуществлялась по одному регистру-модификатору. Однако идею модификации легко обобщить на случай нескольких модификаторов. Для этого надо в командах вместе с адресом указывать несколько таких регистров. В ПК разрешено указывать сразу два модификатора, причем один из них обязательно должен быть регистром BX или BP, а другой – регистром SI или DI (модифицировать по парам BX и BP или SI и DI нельзя). Возможный пример:

```
MOV AX,A[BX][SI]
```

В данном случае исполнительный адрес вычисляется по формуле

$$Аисп = (A + [BX] + [SI]) \bmod 2^{16}$$

Модификация по двум регистрам обычно используется при работе с двумерными массивами. Пусть, к примеру, имеется матрица A размером 10x20:

```
A DB 10 DUP(20 DUP(?)) ;A[0..9,0..19]
```

и требуется записать в регистр AL количество таких строк этой матрицы, в которых начальный элемент строки встречается в ней еще раз.

При расположении элементов матрицы в памяти по строкам (первые 20 байт – начальная строка матрицы, следующие 20 байт – вторая строка и т. д.) адрес элемента A[i,j] равен A+20*i+j. Для хранения величины 20*i отведем регистр BX, а для хранения j – регистр SI. Тогда A[BX] – это начальный адрес i-й строки матрицы, а A[BX][SI] – адрес j-го элемент этой строки.

```
    MOV AL,0        ;количество искомых строк
;внешний цикл (по строкам)
    MOV CX,10       ;счетчик внешнего цикла
    MOV BX,0        ;смещение от A до начала строки (20*i)
 L: MOV AH,A[BX]    ;AH – начальный элемент строки
    MOV DX,CX       ;спасти CX внешнего цикла
;внутренний цикл (по столбцам)
    MOV CX,19       ;счетчик внутреннего цикла
    MOV SI,0        ;индекс элемента внутри строки (j)
L1: INC SI          ;j:=j+1
    CMP A[BX][SI],AH  ;A[i,j]=AH?
    LOOPNE L1       ;цикл, пока A[i,j]<>AH, но не более 19 раз
    JNE L2          ;AH не повторился --> L2
    INC AL          ;учет строки
;конец внутреннего цикла
L2: MOV CX,DX       ;восстановить CX для внешнего цикла
    ADD BX,20       ;на начало следующей строки
    LOOP L          ;цикл 10 раз
```

5.2.5. Запись модифицируемых адресов в ЯА

Уточним правила записи модифицируемых адресов в ЯА.

Пусть A обозначает адресное выражение, а E – любое выражение (адресное или константное), тогда в ЯА допустимы следующие три основные формы записи адресов в командах, которые задают следующие исполнительные адреса:

```
A :          Аисп=A
E[M] :       Аисп=(E+[M]) mod 2^16        (M: BX,BP,SI,DI)
E[M1][M2] : Аисп=(E+[M1]+[M2]) mod 2^16  (M1: BX,BP; M2: SI,DI)
```

Замечание. Если E=0, то 0 можно опустить: 0[M] = [M].

Напомним еще раз, что в ПК в качестве регистра-модификатора можно использовать не любой регистр, а только один из следующих четырех: BX, BP, SI или DI. При модификации только по одному регистру модификатором может быть любой из этих четырех регистров. Однако при модификации по двум регистрам в ПК разрешается указывать не любую пару регистров-модификаторов, а только такую, где один регистр – это BX или BP, а другой регистр – SI или DI.

Замечание. Регистр BP используется обычно для работы со стеком, для доступа к его элементам, о чем будет рассказано в гл. 8. Использовать этот регистр для модификации адресов из других участков памяти нельзя (точнее, можно, но особым образом). Поэтому мы пока забудем про этот модификатор, как будто бы его и нет.

Отметим, что модификация адреса не меняет тип адреса: например, если X – имя байтовой переменной, то TYPE X[SI] = BYTE. Если же перед модификатором указано константное выражение (например, 1[BX]), то тип такого адреса считается неопределенным.

Помимо трех указанных основных способов записи адресов в ЯА допускаются и другие, являющиеся производными от этих трех. Но прежде чем рассмотреть их, опишем принятые в ЯА соглашения об использовании квадратных скобок при записи операндов команд.

1) Запись в квадратных скобках имени регистра-модификатора (BX, BP, SI или DI) эквивалентна выписыванию содержимого этого регистра (тогда как имя регистра без квадратных скобок обозначает сам регистр).

Примеры:

```
 A DW 99
AA DW A
   ...
   MOV BX,AA      ;в BX – адрес A
   MOV CX,[BX]    ;CX:=A (CX:=99)
   MOV CX,BX      ;CX:=BX (CX:=адрес A)
```

Отметим, что заключать в квадратные скобки имена регистров, не являющихся модификаторами (AX, SP, DS, AL и т. п.), нельзя.

2) Любое выражение можно заключить в квадратные скобки, от этого его смысл не изменится (в то же время снятие скобок может изменить смысл).

Примеры:

```
MOV CX,[2]          ;= MOV CX,2
MOV CX,[A]          ;= MOV CX,A
MOV CX,[A+2[BX]]    ;= MOV CX,A+2[BX]
```

3) Следующие записи эквивалентны:

```
[x][y] = [x]+[y] = [x+y]
```

Другими словами, выписывание рядом двух выражений в квадратных скобках означает сумму этих выражений.

Пример:

```
MOV CX,[BX][SI]    ;= MOV CX,[BX]+[SI] = MOV CX,[BX+SI]
```

Так вот, используя эти соглашения вместе со свойством коммутативности сложения и отталкиваясь от трех основных способов записи адресов в командах, можно получить новые варианты записи адресов. Например, в каждой из следующих строк указаны эквивалентные формы записи одного и того же адреса:

```
A+1, [A+1], [A]+[1], [A][1], A[1], 1[A], [A]+1, ...
5[SI], [5][SI], [5]+[SI], [SI+5], [SI]+5, ...
A-2[BX], [A-2]+[BX], [A-2+BX], A[BX-2], A[BX]-2, ...
A[BX][DI], A[BX+DI], [A+BX+DI], A[BX]+[DI], ...
0[BX][SI], [BX][SI], [BX]+[SI], [BX+SI], [SI][BX], ...
```

Итак, в ЯА один и тот же адрес можно записать разными способами. Однако при этом следует помнить, что используемая запись должна быть эквивалентной одной из трех основных форм записи таких адресов. В частности, в записи не должно быть:

- суммы двух адресов (A+B);

- суммы, одним из слагаемых которой является регистр (BX+2);
- запрещенных сочетаний регистров ([SI+DI]);
- регистров, не являющихся модификаторами (A[CX], 2[BL]);
- имени или числа непосредственно за [] ([SI]5, [BX]A).

Кроме того, адрес не должен сводиться к числу ([5]), т. к. тогда это будет константное выражение, а не адресное.

В остальном ограничений нет, и каждый выбирает ту форму записи, что ему больше нравится. Мы в дальнейшем будем использовать форму, в которой до скобок ставится имя переменной, а остальные слагаемые указываются в квадратных скобках (например, A[SI+3]). Такие записи похожи на привычные обозначения переменных с индексами в языках высокого уровня (A[i+3]).

И еще одно замечание. Адресные выражения с модификаторами можно использовать при записи операндов команд и некоторых директив (EQU, PTR и др.), но ни в коем случае нельзя указывать в директивах определения данных. Например, директива

```
X DW 1[SI]
```

является ошибочной, т. к. ассемблер обязан вычислить ее операнд еще на этапе трансляции программы, чтобы подставить его значение в ячейку X, но в это время значение регистра SI, конечно, неизвестно.

5.3. Команды LEA и XLAT

5.3.1. Команда LEA

При использовании регистров-модификаторов часто приходится записывать в них те или иные адреса. Пусть, к примеру, нам надо занести в регистр BX адрес переменной X:

```
X DW 88
```

Чтобы сделать это, можно завести еще одну переменную, значением которой является адрес X:

```
Y DW X
```

а затем переслать значение этой переменной в регистр BX:

```
MOV BX,Y
```

Но ясно, что это несколько искусственный способ загрузки адреса в регистр. В то же время такое действие встречается довольно часто в реальных программах. Поэтому в систему команд ПК введена специальная команда такой загрузки:

Загрузка исполнительного адреса (load effective address): `LEA r16,A`

Эта команда вычисляет исполнительный адрес второго операнда и записывает его в регистр r16: r16:=Аисп. Флаги команда не меняет. В качестве

первого операнда может быть указан любой регистр общего назначения, а в качестве второго – любое адресное выражение (с модификаторами или без них).

Одним из примеров, где полезна команда LEA, является вывод строки по опе-рации OUTSTR. Эта операция, напомним, требует, чтобы начальный адрес выводимой строки находился в регистре DX. Так вот, засылку этого адреса в DX и следует делать по команде LEA:

```
S DB 'a+b=c','$'
  ...
  LEA DX,S          ;DX:=адрес S
  OUTSTR            ;будет выведено: a+b=c
```

Рассмотрим особенности команды LEA. При этом будем предполагать, что в программе имеются следующие описания:

```
Q DW 45
R DW -8
```

Прежде всего отметим, что команда LEA очень похожа на команду MOV, но между ними имеется и принципиальное различие: если LEA записывает в регистр сам адрес, указанный в команде, то MOV записывает содержимое ячейки с этим адресом:

```
LEA BX,Q   ;BX:=адрес Q
MOV BX,Q   ;BX:=содержимое Q (=45)
```

В команде LEA второй операнд может быть любым адресом – и адресом байта, и адресом слова и т. д. Однако в качестве этого операнда нельзя указывать константное выражение или имя регистра:

```
LEA CX,88   ;ошибка
LEA CX,BX   ;ошибка
```

Если в качестве второго операнда указан модифицируемый адрес, то сначала вычисляется исполнительный адрес и лишь затем происходит загрузка в регистр:

```
MOV SI,2
LEA AX,Q[SI]   ;AX:=Аисп=Q+[SI]=адрес(Q+2)=адрес(R)
```

В частности, этим можно воспользоваться для пересылки в какой-либо регистр значения регистра-модификатора, увеличенного или уменьшенного на некоторое число:

```
MOV BX,50
LEA CX,[BX+2]  ;CX:=[BX]+2=50+2=52
LEA DI,[DI-3]  ;DI:=DI-3 (но понятнее: SUB DI,3)
```

5.3.2. Команда XLAT

Рассмотрим еще одну команду, связанную с модификацией адресов. Это так называемая команда перевода, перекодировки (правда, не очень понятно, откуда в ее мнемокоде взялась буква X):

```
Перекодировка (translate): XLAT
```

Действие этой команды заключается в том, что содержимое байта памяти, адрес которого равен сумме текущих значений регистров BX и AL, записывается в регистр AL: AL:=байт по адресу [BX+AL]. Флаги не меняются.

Команда XLAT используется для перекодировки символов. Предполагается, что имеется таблица (байтовый массив) размеров до 256 байт, i-й элемент которой трактуется как новый код символа с кодом i. Начальный адрес этой таблицы должен находиться в регистре BX, а исходный код (i) перекодируемого символа – в регистре AL. Команда присваивает регистру AL новый код символа, взятый из i-го элемента таблицы.

В качестве примера рассмотрим, как с помощью этой команды можно преобразовать любую шестнадцатеричную цифру-число (от 0 до 15) в цифру-символ. Для этого определим в программе таблицу DIG16, перечислив в ней все символы, изображающие шестнадцатеричные цифры:

```
DIG16 DB '0123456789ABCDEF' ;DIG16[0..15]
```

Пусть в регистре AL находится число от 0 до 15 (скажем, 14). Тогда записать в этот регистр соответствующий символ-цифру ('E') можно так:

```
LEA BX,DIG16 ;BX – на начало DIG16
XLAT         ;AL:=DIG16[AL]
```

5.4. Структуры

Мы рассмотрели, как на ЯА описываются и обрабатываются массивы. Теперь рассмотрим реализацию другого составного типа данных, который в языке Паскаль называют записью. Сразу отметим, что в ЯА записи принято называть структурами, поэтому в дальнейшем мы будем называть их только структурами.

5.4.1. Описание типа структуры

Структура – это составной объект, занимающий несколько соседних ячеек памяти. Компоненты структуры называются полями, они могут быть разного типа (размера): например, одно поле может быть байтом, другое – словом и т. д. Поля именуются, доступ к полям осуществляется по именам.

Прежде чем использовать структуру, надо описать ее тип – указать, сколько в ней полей, каковы имена у полей и т. д. Описание типа структуры выглядит так:

```
<имя типа> STRUC
           <описание поля>
           ...
           <описание поля>
<имя типа> ENDS
```

Описание типа открывает директива STRUC (structure), где указывается имя, которое мы дали типу структуры. Это же имя обязательно должно быть повторено в директиве ENDS (end of structure), оканчивающей описание типа. Между этими двумя директивами может быть указано любое число директив,

описывающих поля структуры. Каждое поле описывается одной директивой, причем это обязательно должна быть директива определения данных DB, DW или DD. Имя, указанное в такой директиве, считается именем соответствующего поля структуры.

Например, тип структуры DATE (дата) из трех полей Y (year, год), M (month, месяц) и D (day, день) можно описать так:

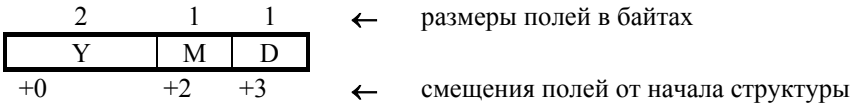
```
DATE STRUC
   Y DW 1994
   M DB 3
   D DB ?
DATE ENDS
```

Описание типа структуры носит чисто информационный характер, по нему ассемблер ничего не заносит в машинную программу, поэтому такое описание можно размещать в любом месте программы, но обязательно до описания переменных этого типа.

Отметим, что, в отличие от языка Паскаль, в ЯА имена полей не локализуются в структуре и потому должны быть уникальными, не должны совпадать с именами других объектов программы. Кроме того, поля структуры не могут быть снова структурами, т. е. в ЯА не допускается вложенность структур.

И еще одно отличие от Паскаля: в ЯА полям структуры можно давать так называемые значения по умолчанию. Если в правой части директивы, описывающей поле, указан знак неопределенного значения (?), то это поле не будет иметь никакого значения по умолчанию, но если здесь указана иная величина, то она и будет значением этого поля по умолчанию. В нашем примере значением поля Y по умолчанию будет число 1994, поля M – число 3, а у поля D нет значения по умолчанию. Как используются эти значения по умолчанию, мы увидим чуть позже.

В ЯА имя любого поля структуры относится к простейшим константным выражениям, его значением является смещение данного поля относительно начала структуры. При выделении памяти под переменную данного типа ассемблер размещает ее поля в соседних ячейках памяти. Например, под любую переменную типа DATE будут выделены 4 соседних байта, причем два первых – это поле Y, следующий байт – это поле M и последний байт – поле D:



Начало поля Y совпадает с началом всей структуры, поэтому имени Y приписывается значение 0; поле M сдвинуто от начала структуры на 2 байта, поэтому имени M приписывается значение 2; имя D получает значение 3. Встречая затем в программе имена полей, ассемблер будет заменять их на эти значения. Зачем все это нужно, мы увидим позже.

5.4.2. Описание переменных-структур

После того как описан тип структуры, можно определять в программе переменные этого типа, отводить под них память (напомним: под описание типа память не отводится). Такие переменные называются переменными-структурами или просто структурами. Описываются они с помощью директив следующего вида:

```
имя_переменной имя_типа <нач_знач {, нач_знач}>
```

(здесь уголки – не метасимволы, а явно указываемые символы, поэтому, чтобы не было путаницы, мы не заключаем метапеременные в уголки), причем каждое нач_знач – это ?, выражение, строка или "пусто".

Примеры описания переменных-структур:

```
DT1 DATE <?,6,9>
DT2 DATE <1998,,>
DT3 DATE <,,>
```

	Y	M	D
DT1	?	6	9
DT2	1998	3	?
DT3	1994	3	?

Отметим, что это директивы особого рода. До сих пор мы рассматривали директивы, названиями которых были служебные слова, т. е. предопределенные имена, а здесь названием является имя DATE, которое мы придумали сами.

Каждая такая директива описывает одну переменную, имя которой указывается в начале директивы. Для каждой переменной в памяти выделяется 4 байта, причем два первых из них считаются полем Y этой переменной, третий байт – полем M и последний байт – полем D.

Кроме того, по этим директивам полям переменных даются начальные значения – они указываются в уголках через запятую. При этом i-е нач_знач соответствует i-му полю. Указанное начальное значение будет записано в соответствующую ячейку структуры; позже, при выполнении программы, это значение можно и изменить, но вначале у поля будет именно это значение. Естественно, начальное значение должно умещаться в размер поля, иначе будет зафиксирована ошибка.

Правила задания начальных значений для полей следующие:

- если в качестве начального значения указан знак ?, то соответствующее поле не получит никакого начального значения (таково, например, поле Y в переменной DT1);
- если в качестве начального значения указано выражение или строка, то значение этого выражения или сама строка становится начальным значением поля (таковы, например, поля M и D в переменной DT1 и поле Y в переменной DT2);
- если же начальное значение не указано (указано "пусто"), то возможны два случая:

- если при описании типа структуры в этом поле указано какое-то значение по умолчанию, то оно и становится начальным значением этого поля данной переменной (таково, например, поле M в переменных DT2 и DT3);
- если же в описании типа для этого поля не указано значения по умолчанию, то это поле данной переменной не получит никакого начального значения (таково, например, поле D в переменной DT3).

Короче говоря, смысл этих правил следующий: если при описании переменной для поля указано ? или какое-то значение, то эта величина и берется в качестве начального значения поля (значение по умолчанию в данном случае полностью игнорируется), а если не указано ничего, то берется значение из описания типа.

Зачем нужны значения по умолчанию? Практика показывает, что довольно часто у разных структур одного и того же типа некоторое поле должно иметь одно и то же начальное значение. Например, может быть так, что у большинства переменных-дат поле "год" должно равняться 1994. Так вот, чтобы такое часто встречающееся значение не выписывать каждый раз заново при описании каждой переменной, это значение указывают только раз – при описании типа этих переменных и договариваются, что если при описании переменной не указано иного значения для этого поля, то берется в расчет именно это значение по умолчанию.

Отметим, что при задании начальных значений полей возможно следующее сокращение: если в уголках не указываются начальные значения для нескольких последних полей, т. е. в конце стоит несколько запятых подряд, то эти запятые можно опустить. Например:

```
DT2 DATE <1998,,> эквивалентно DT2 DATE <1998>
DT3 DATE <,,>     эквивалентно DT3 DATE <>
```

(уголки опускать нельзя). Однако в начале или в середине запятые опускать нельзя. Например, <,,8> нельзя записать как <8>, поскольку последняя конструкция является сокращением для <8,,>.

И, наконец, еще одно замечание об описании переменных-структур. В ЯА одной директивой можно описать сразу несколько структур, т. е. можно описать массив, элементами которого являются структуры. Для этого в директиве указывается несколько операндов и/или конструкция повторения DUP. Например, по директиве

```
DTS DATE <,12,5>, 10 DUP (<>)
```

описывается массив из 11 структур типа DATE, причем поля первой из них будут иметь следующие начальные значения: 1994, 12 и 5, а остальные 10 структур получают один и тот же набор начальных значений, взятых по умолчанию: 1994, 3 и ?. При этом имя DTS получает только первая из 11 описанных структур, остальные же остаются безымянными и доступ к ним осуществляется по адресным выражениям вида DTS+4, DTS+8 и т. п.

5.4.3. Ссылки на поля структур

Описав тип структуры и переменные этого типа, мы получаем право работать с этими переменными-структурами. Как единое целое структуры обрабатываются довольно редко, обычно они обрабатываются по полям. Так вот, чтобы сослаться на поле структуры, надо использовать конструкцию вида

```
<имя переменной-структуры>.<имя поля>
```

аналогичную обозначению поля в языке Паскаль.

Возможный пример: `DT3.D`

Такая конструкция обозначает ту ячейку памяти, которую занимает указанное поле указанной переменной. Встречая эту конструкцию, ассемблер заменяет ее на адрес данной ячейки. При этом тип (размер) этого адреса считается равным типу данного поля, например: TYPE DT2.Y = WORD.

Рассмотрим такую задачу: если в переменной DT1 хранится мартовская дата, то требуется записать сюда дату следующего дня года. Решение этой задачи на ЯА записывается так:

```
      CMP DT1.M,3
      JNE FIN       ;не март -> fin
      CMP DT1.D,31
      JE APR1       ;31 марта -> apr1
      INC DT1.D     ;следующий день в марте
      JMP FIN
APR1: MOV DT1.M,4   ;замена 31 марта на 1 апреля
      MOV DT1.D,1
FIN:  ...
```

5.4.4. Уточнения

Мы рассказали об основных средствах ЯА для работы со структурами. Теперь рассмотрим некоторые дополнительные возможности и кое-что уточним.

Тип имени структуры

Ассемблер приписывает имени типа структуры и имени структуры тип (размер), равный числу байтов, занимаемых этой структурой. Например:

```
TYPE DATE = TYPE DT1 = TYPE DT2 = TYPE DTS = 4
```

В связи с этим присваивание DT2:=DT1, если нежелательно по какой-либо причине явно указывать размер этих структур, можно реализовать так:

```
   MOV CX,TYPE DATE          ;CX:=размер структур типа DATE
   MOV SI,0                  ;i:=0
L: MOV AL,BYTE PTR DT1[SI]   ;побайтовая пересылка DT1 => DT2
   MOV BYTE PTR DT2[SI],AL
   INC SI                    ;i:=i+1
   LOOP L                    ;цикл CX раз
```

Отметим, что оператор PTR здесь обязателен, т. к. размеры операндов AL и DT1[SI] (т. е. 1 и 4) различны.

Оператор «.»

Точка, указываемая в ссылке на поле структуры, – это на самом деле один из операторов ЯА, обращение к которому в общем случае имеет такой вид:

```
<адресное выражение>.<имя поля структуры>
```

Этот оператор относится к адресным выражениям и обозначает адрес, вычисляемый по формуле:

```
<адресное выражение>+<смещение поля в структуре>
```

причем тип этого адреса совпадает с типом (размером) указанного поля.

Например:

```
DT1.D = DT1+D = DT1+3
TYPE (DT1.D) = TYPE D = BYTE
```

(напомним, что имя поля заменяется ассемблером на смещение поля относительно начала структуры).

В общем случае адресное выражение может не иметь никакого отношения к структурам, к указанному полю. Например, если имеется описание

```
X DW ?
```

тогда допустима запись X.M, хотя между X и M и нет никакой связи; обозначает же эта запись адрес X+M=X+2. Однако увлекаться такой возможностью не следует. Чтобы ссылка на поле была осмысленной, адресное выражение, конечно, должно указывать на структуру. Причем это выражение может быть любой сложности, например:

```
MOV AX,(DTS+8).Y
MOV SI,8
INC (DTS[SI]).M ;Аисп = (DTS+[SI]).M = (DTS+8).M
LEA BX,DT1
MOV [BX].D,10;Аисп = [BX]+D = DT1.D
```

Здесь, правда, надо учитывать следующее: если такое адресное выражение – не имя и не косвенная ссылка (вида [BX]), то его надо заключать в круглые скобки. Дело в том, что значение (адрес) ссылки на поле будет вычислено правильно и без скобок, но вот тип этого адреса (по непонятной причине) будет определен ассемблером неправильно. Так, записи (DTS[SI]).M и DTS[SI].M задают один и тот же адрес DTS+[SI]+M, но в первом случае ассемблер приписывает этому адресу тип имени M (т. е. 1), а во втором – тип имени DTS (т. е. 4).

Нескалярные поля структур

Пока мы рассматривали лишь случаи, когда поля структур были скалярными, т. е. представляли одну величину. Но, как мы знаем, в директивах DB, DW и DD можно указывать несколько операндов и конструкцию повторения

DUP. Подобные директивы можно использовать и при описании полей структур. Например, допустимо такое описание типа структуры:

```
 STUD STRUC          ;студент
  FAM DB 10 DUP(?)   ;фамилия
 NAME DB '****'      ;имя
   GR DW ?           ;группа
MARKS DB 5,5,5       ;оценки
      ENDS
```

Здесь на поле FAM отводится 10 байт, на NAME – 4 байта, на GR – 2 байта и на MARKS – 3 байта.

Так вот, если в директиве, описывающей поле, имеется более одного операнда или конструкция повторения, то при описании переменной данного типа для этого поля нельзя указывать начальное значение, нельзя указывать и знак "?" – соответствующая позиция в уголках должна быть пустой. Из этого правила есть только одно исключение: если поле описано как строка, то такому полю можно давать начальное значение, но оно обязано быть также строкой, причем равной или меньшей длины (при меньшей длине добавляются пробелы справа).

Примеры:

```
S1 STUD <'Иванов', ...>  ; нельзя (поле FAM - не строка)
S2 STUD <,'Оля',101,>    ; можно (S2.NAME='Оля ', S2.GR=101)
S3 STUD <,'Ольга'>       ; нельзя (5 символов вместо 4)
```

Причина подобных ограничений в том, что если разрешить задавать начальные значения для нескалярных полей, то будет путаница в том, какое значение к какому полю относится.

5.5. Примеры

Здесь рассматриваются некоторые примеры на обработку массивов и структур, на использование индексных регистров и приводятся с пояснениями решения этих примеров.

Пример 1

```
N EQU 100
X DW N DUP(?);X[0..N-1]
```

Рассматривая элементы массива X как числа со знаком, записать в регистр AL индекс (от 0 до N-1) максимального элемента массива (первого, если таких элементов несколько).

Решение

Поскольку адрес(X[i])=X+2*i, то в регистре (скажем, SI), используемом как индексный, приходится хранить удвоенный индекс (2*i). Пока идет поиск максимального элемента и его индекса, будем работать с удвоенными индексами, и только в конце разделим удвоенный индекс максимального элемента на 2, чтобы получить "настоящий" индекс.

```
      MOV BX,X      ;BX:=X[0] (начальный max)
      MOV AX,0      ;AX:=2*(индекс max)
      MOV SI,2      ;SI:=2*i (i – индекс очередного элемента)
      MOV CX,N-1    ;счетчик цикла
 MAX: CMP X[SI],BX  ;X[i]=BX?
      JLE MAX1      ;при <= перейти на MAX1
      MOV BX,X[SI]  ;запомнить новый max
      MOV AX,SI     ;и его удвоенный индекс
MAX1: ADD SI,2      ;i:=i+1
      LOOP MAX      ;цикл CX раз
      MOV CL,2
      DIV CL        ;AL:=AX div 2 (настоящий индекс max)
```

Пример 2

```
N EQU 50
Y DB N DUP(?) ;Y[0..N-1]
```

Требуется циклически сдвинуть элементы массива Y на две позиции вперед:

```
(Y[0],Y[1],Y[2],...,Y[N-1]) --> (Y[2],...,Y[N-1],Y[0],Y[1])
```

Решение

Здесь нужно в цикле выполнять операцию Y[i]:=Y[i+2] и тем самым одновременно ссылаться на соседние элементы массива Y при условии, что в индексном регистре (скажем, в DI) находится индекс (i) одного из них. Можно, конечно, перед ссылкой на элемент Y[i+2] изменить соответствующим образом значение этого регистра (записать i+2), а после восстановить его прежнее значение (i), однако проще поступить иначе. Поскольку

```
адрес(Y[i]) = Y+i
адрес(Y[i+2]) = Y+(i+2) = (Y+2)+i,
```

то при DI=i ссылка на Y[i] задается адресным выражением Y[DI], а ссылка на Y[i+2] – выражением Y+2[DI] или Y[DI+2]. Таким образом, "добавку" 2 проще указывать в самой команде, чем учитывать ее в индексном регистре.

```
       MOV AH,Y       ;спасти Y[0] и Y[1]
       MOV AL,Y+1
       MOV DI,0       ;i:=0
       MOV CX,N-2     ;счетчик цикла (число сдвигов вперед)
SHIFT: MOV BH,Y[DI+2]
       MOV Y[DI],BH   ;Y[i]:=Y[i+2]
       INC DI         ;i:=i+1
       LOOP SHIFT
       MOV Y+N-2,AH   ;Y[n-2]:=Y[0]
       MOV Y+N-1,AL   ;Y[n-1]:=Y[1]
```

Пример 3

```
N EQU 65
S DB N DUP(?);S[0..N-1]
```

Определить, симметричен ли массив S, т. е. равны ли его элементы, равноудаленные от концов, и записать ответ 1 (симметричен) или 0 в регистр AL.

Решение.

Здесь надо в цикле выполнять сравнение элементов S[i] и S[N-1-i] при значениях i от 0 до (N div 2). Конечно, имея индекс i для доступа к первому из этих элементов, можно получить и индекс N-1-i для доступа ко второму элементу, однако это потребует нескольких команд. Проще считать величину N-1-i новым индексом (j) и работать сразу с двумя индексами, увеличивая i на 1 и одновременно уменьшая j на 1.

```
     MOV AL,0
     MOV SI,0        ;i:=0
     MOV DI,N-1      ;j:=n-1
     MOV CX,N/2      ;число сравниваемых пар
SYM: MOV AH,S[SI]
     CMP AH,S[DI]    ;S[i]=S[j]?
     JNE FIN         ;не равно --> выход с AL=0
     INC SI          ;i:=i+1
     DEC DI          ;j:=j-1
     LOOP SYM
     MOV AL,1        ;равны все пары -> AL:=1
FIN:
```

Пример 4

Для ввода задана последовательность больших латинских букв, за которой следует точка. Вывести в алфавитном порядке все различные буквы, входящие в эту последовательность.

Решение

Прежде всего опишем в программе нулевой байтовый массив LET из 26 элементов, первый из которых поставим в соответствие букве A, второй – букве B, ..., последний – букве Z. Буквы будем вводить по очереди и для каждой из них в соответствующий ей элемент массива будем записывать 1. По окончанию ввода в массиве LET будут помечены единичками все буквы, которые хотя бы раз входили в исходную последовательность. В конце просматриваем от начала этот массив и, если очередной его элемент равен 1, выводим соответствующую букву.

Отметим, что если код введенной буквы записать в регистр BX, тогда индекс элемента массива LET, соответствующего этой букве, равен разности значения BX и кода буквы A, поэтому ссылка на этот элемент задается выражением LET[BX-'A'].

```
   LET DB 26 DUP(0)        ;LET['A'..'Z']
PROMPT DB 'Вводите буквы: ','$'
       ...
;ввод букв и заполнение массива LET
       LEA DX,PROMPT       ;приглашение к вводу
       OUTSTR
       MOV BH,0            ;нужно для расширения BL до BX
   IN: INCH BL             ;BL (BX) – код очередной буквы
       CMP BL,'.'
       JE OUT              ;точка --> на вывод
```

```
       MOV LET[BX-'A'],1  ;запомнить, что буква была
       JMP IN
;вывод по алфавиту букв, которые входили в исходный текст
  OUT: MOV CX,26          ;длина массива LET
       MOV BX,'A'         ;BX (BL) – коды букв от A до Z
 OUT1: CMP LET[BX-'A'],1  ;1 в соответствующем элементе LET?
       JNE OUT2
       OUTCH BL           ;при 1 вывести букву
 OUT2: INC BX
       LOOP OUT1
       NEWLINE
```

Пример 5

Пусть FROM и TO – переменные размером в слово, значениями которых являются некоторые адреса. Требуется переписать (скопировать) 45 байт начиная с адреса, который находится в FROM, в область памяти, начальный адрес которой указан в TO.

Решение

Обозначим адрес из FROM как f, а адрес из TO – как t. Тогда в задаче нужно осуществить пересылку байтов из ячеек с адресами f+i в ячейки с адресами t+i, где i меняется от 0 до 44. Эти адреса не известны заранее и к тому же меняются, поэтому их можно хранить только в регистрах. В каких? Здесь возможны два варианта.

Во-первых, можно сначала записать, скажем, в регистры SI и DI адреса f и t, а затем увеличивать их на 1:

```
      MOV SI,FROM   ;SI=f+i (пока i=0)
      MOV DI,TO     ;DI=t+i (пока i=0)
      MOV CX,45     ;число пересылаемых байтов
COPY: MOV AH,[SI]
      MOV [DI],AH   ;f+i --> t+i
      INC SI
      INC DI
      LOOP COPY
```

Здесь мы в регистрах SI и DI хранили полные адреса f+i и t+i. Однако эти адреса можно рассматривать и как состоящие из двух частей, каждую из которых можно хранить в отдельном регистре. Например, f можно хранить в регистре SI, t – в DI, i – в BX. Тогда полные адреса f+i и t+i должны задаваться выражениями [SI+BX] и [DI+BX], причем значения регистров SI и DI не будут меняться, а будет меняться только значение регистра BX:

```
      MOV SI,FROM       ;SI=f
      MOV DI,TO         ;DI=t
      MOV BX,0          ;BX=i (пока 0)
      MOV CX,45         ;число пересылаемых байтов
COPY: MOV AH,[SI+BX]
      MOV [DI+BX],AH    ;f+i --> t+i
      INC BX
      LOOP COPY
```

Какой из этих двух вариантов лучше? Однозначного ответа нет, т. к. в первом варианте используется меньше регистров, а во втором – меньше команд выполняется в цикле. Правда, во втором варианте надо быть внимательным при распределении регистров-модификаторов, чтобы не попасться на одновременную модификацию по регистрам SI и DI: если бы мы выбрали для хранения величины i регистр SI, тогда адрес f или t пришлось бы хранить в регистре DI и потому пришлось бы использовать выражение [DI+SI], что недопустимо.

Пример 6

Пусть в программе описан структурный тип STUD (студент):

```
 STUD STRUC
  FAM DB 20 DUP(?)    ;фамилия
   GR DW ?            ;номер группы
MARKS DB 5 DUP(?)     ;оценки
 STUD ENDS
```

и пусть в массиве

```
S STUD 300(<>)
```

собрана информация о 300 студентах. Требуется подсчитать и записать в регистр DX количество студентов-отличников.

Решение

Алгоритм решения этой задачи очевиден: просматриваем по очереди всех студентов, для каждого из них сравниваем его оценки с 5 и, если все оценки отличные, увеличиваем счетчик DX на 1. Основная проблема здесь – как "добраться" до очередного студента и его оценок?

В регистре BX будем хранить смещение от начала массива S до начала его очередного элемента (до информации об очередном студенте), в связи с чем выражение S[BX] будет означать начальный адрес этого элемента. Этот элемент является структурой, поэтому доступ к его полю MARKS будем осуществлять с помощью конструкции (S[BX]).MARKS. Это поле, в свою очередь, является массивом. Для хранения индекса очередного элемента этого массива (очередной оценки) будем использовать регистр SI. Следовательно, выражение (S[BX]).MARKS[SI] будет означать адрес байта, где находится очередная оценка. Отметим, что последнее выражение вполне корректно, т. к. сводится к адресу S[BX+22+SI], где 22 – значение имени MARKS, и имеет тип BYTE.

С учетом всего сказанного получается такое решение задачи:

```
      MOV DX,0          ;число отличников
;внешний цикл (по студентам)
      MOV CX,300        ;общее число студентов
      MOV BX,0          ;начало информации об очередном
                        ;студенте
```

```
  LS: MOV AX,CX          ;спасти CX
;внутренний цикл (по оценкам)
      MOV CX,5           ;число оценок
      MOV SI,0           ;индекс оценки
  LM: CMP (S[BX]).MARKS[SI],5   ;оценка=5?
      JNE NEXT           ;нет -> NEXT
      INC SI
      LOOP LM
      INC DX             ;учет отличника
;к следующему студенту
NEXT: MOV CX,AX          ;восстановить CX
      ADD BX,TYPE STUD   ;на начало информации о след. студенте
      LOOP LS
```

6. БИТОВЫЕ ОПЕРАЦИИ. УПАКОВАННЫЕ ДАННЫЕ

В этой главе описываются команды ПК, которые рассматривают свои операнды просто как последовательности битов; это команды, реализующие логические операции, и команды сдвига, осуществляющие перемещение битов операнда на несколько позиций влево или вправо. Рассматривается также применение этих команд к обработке упакованных данных.

6.1. Логические команды

Логическим командам, которые, как следует из их названия, выполняют логические операции – отрицание, конъюнкцию и дизъюнкцию, присущ ряд общих черт.

Во-первых, все они реализуют, как говорят, поразрядные операции. Это означает, что i-й разряд результата зависит только от i-ых разрядов операндов и ни от чего иного. При этом одна и та же операция выполняется сразу над всеми разрядами операндов одновременно, параллельно.

Во-вторых, во всех этих командах бит 1 трактуется как "истина", а бит 0 – как "ложь". Именно при такой трактовке эти команды и реализуют логические операции отрицания, конъюнкции и дизъюнкции.

В-третьих, эти команды меняют все флаги условий, но интерес обычно вызывает только флаг нуля ZF, который, напомним, принимает значение 1, если получился нулевой результат, и равен 0, если в результате есть хотя бы одна единичка. Что касается других флагов, то они, предназначенные для работы с числами, в логических операциях малоинформативны.

В-четвертых, операндами логических команд должны быть либо байты, либо слова, но не то и другое одновременно.

А теперь рассмотрим сами логические команды.

Отрицание: NOT op

Допустимые типы операнда: r8, m8, r16, m16.

Эта команда меняет значение каждого бита операнда на противоположное: 0 на 1 и 1 на 0; результат записывается на место операнда. Например:

```
MOV AL,1100b    ;AL=00001100b
NOT AL          ;AL=11110011b
```

Конъюнкция (логическое умножение): AND op1,op2

В этой команде допустимы следующие комбинации операндов:

op1	*op2*
r8	i8, r8, m8
m8	i8, r8
r16	i16, r16, m16
m16	i16, r16

Команда производит поразрядное логическое умножение операндов и записывает результат на место первого операнда. i-й бит результата равен 1, только если i-е биты обоих операндов равны 1, и равен 0, если хотя бы в одном операнде i-й бит нулевой (см. таблицу). Например:

```
MOV AL,1100b ;AL=00001100b
AND AL,1010b ;AL=00001000b
```

x	*y*	*x and y*	*x or y*	*x xor y*
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Проверка: TEST op1,op2

Это аналог команды AND, но результат логического умножения никуда не записывается. Главное в команде TEST – установка флагов. Как уже было сказано, в логических командах интерес представляет только флаг нуля ZF. Так вот, он равен 1, если в результате логического умножения получился нулевой ответ, и равен 0, если в ответе есть хотя бы одна двоичная 1.

Примеры:

```
MOV BH,1100b
TEST BH,0011b ;= 00000000b -> ZF=1
TEST BH,1010b ;= 00001000b -> ZF=0
```

Команда TEST обычно применяется для проверки, являются ли нулевыми какие-то разряды операнда. Например, сделать переход на метку L в случае, если правые три бита регистра AX нулевые, можно так:

```
TEST AX,111b
JZ L
```

Дизъюнкция (логическое сложение): OR op1,op2

Допустимые типы операндов – как в команде AND.

Команда производит поразрядное логическое сложение и записывает результат на место первого операнда. Здесь i-й бит результата равен 1, если i-й бит хотя бы одного операнда равен 1, и равен 0 только в одном случае, когда i-е биты обоих операндов нулевые (см. таблицу). Например:

```
MOV CL,1100b
OR CL,1010b  ;CL=00001110b
```

Исключающее ИЛИ (eXclusive OR): XOR op1,op2

Допустимые типы операндов – как в команде AND.

И здесь результат записывается на место первого операнда, причем i-й бит результата равен 0, если i-е биты операндов совпадают, и равен 1, если эти биты различны (см. таблицу). Например:

```
MOV CL,1100b
XOR CL,1010b ;CL=00000110b
```

В обычной речи эта операция соответствует фразе "или то, или другое, но не то и другое одновременно", т. е. исключается случай, когда оба операнда истинны. Отсюда и название "исключающее ИЛИ".

Отметим одну особенность операции XOR: когда операнды команды XOR совпадают, результатом будет нулевое слово. Например, при любом значении AX имеем:

```
XOR AX,AX ;AX:=0
```

Этот прием часто используется для обнуления регистров; по сравнению с другими приемами (MOV AX,0 или SUB AX,AX) он выполняется быстрее.

Вычисление логических выражений

Теперь рассмотрим вполне естественное применение логических команд – вычисление логических выражений.

Здесь есть одна проблема, она связана с машинным представлением логических значений "истина" и "ложь". Дело в том, что в ЭВМ, как правило, нет стандартного представления логических величин. В ПК минимальная порция данных, обрабатываемая командами, – это байт, в связи с чем на одну логическую переменную обычно и отводят один байт. Но с другой стороны, для представления логического значения достаточно всего одного бита. Возникает вопрос: как заполнить 8 бит байта, если нужен только один бит? Это можно сделать по-разному, каждый может выбрать свой способ заполнения байта, поэтому и нет какого-то одного, стандартного способа представления логических величин.

Конечно, выбранное представление должно быть таким, чтобы логические команды правильно реализовывали логические операции, а этого, оказывается, не всегда удается достичь. Например, при, казалось бы, естественном представлении значения "ложь" нулевым байтом, а значения "истины" – байтом с 1 в самом правом разряде:

```
ложь – 0000 0000b, истина – 0000 0001b,
```

команда NOT будет неправильно реализовывать операцию отрицания:

```
MOV DH,1b  ;DH:=true
NOT DH     ;DH:=11111110b <> false
```

Несложно проверить, что "хорошим" является такое представление:

```
"ложь" – 00000000b (00h)
"истина" – 11111111b (0FFh = -1)
```

В этом случае все логические команды будут правильно реализовывать логические операции. Обычно это представление и используется на практике.

Если воспользоваться таким представлением, тогда с программированием вычислений по логическим формулам не будет никаких проблем. Например, если переменные

```
A DB ?
B DB ?
```

трактуются как логические, то вычисление

```
a:=(not a) and b
```

реализуется с помощью следующих команд:

```
MOV AL,A
NOT AL
AND AL,B
MOV A,AL
```

Однако на практике подобные вычисления встречаются крайне редко. Дело в том, что логические выражения обычно используются для записи условий в условных операторах и циклах, а в этих случаях можно обойтись и без логических команд, достаточно лишь команд сравнения и условных переходов. Например, условный оператор

```
if (X>0) or (Y=1) then goto L
```

можно было бы запрограммировать так: сначала полностью вычислить логическое выражение между if и then, а затем проверить, что получилось – истина или ложь. Однако здесь можно поступить и проще: надо сравнивать X с 0 и Y с 1 и тут же с помощью команд условного перехода реагировать на эти проверки:

```
CMP X,0
JG L       ;X>0 -> L
CMP Y,1
JE L       ;Y=1 -> L
```

Поэтому в реальных программах логические команды редко используются для вычисления логических выражений. Намного чаще они используются для работы с упакованными данными, которые будут рассмотрены ниже.

6.2. Команды сдвига

Все команды сдвига имеют два операнда. Первый из них (байт или слово) рассматривается просто как набор битов, которые будут сдвигаться на несколько позиций влево или вправо. Второй же операнд рассматривается как целое без знака и определяет, на сколько разрядов надо сдвинуть первый операнд. Результат сдвига записывается на место первого операнда.

В ПК каждая команда сдвига имеет две разновидности, которые на ЯА записываются следующим образом:

```
<мнемокод> op,1   ;сдвиг op на 1 разряд
<мнемокод> op,CL  ;сдвиг op на CL разрядов (CL>=0)
```

Допустимые типы операнда op: r8, m8, r16, m16.

В первом варианте op сдвигается только на 1 разряд, а во втором варианте op можно сдвигать на любое число разрядов, причем это число должно находиться в байтовом регистре CL и оно всегда трактуется как неотрицательное. (Процессоры 80386 и 80486 в качестве величины сдвига берут 3 правых бита регистра CL при сдвиге байта и 4 правых байта при сдвиге слова.)

Отметим, что второй вариант – это повторение первого варианта CL раз, причем при CL=0 сдвига нет. Само значение CL не меняется командой сдвига. С учетом этого в дальнейшем ради краткости будем описывать действия команд сдвига только для первого варианта.

Команды сдвига меняют все флаги, но, как правило, интерес представляет только флаг переноса CF.

Все команды сдвига можно разделить на команды логического сдвига (или просто сдвига), арифметического и циклического.

6.2.1. Логические сдвиги

В двух командах, называемых логическими сдвигами, в сдвиге участвуют все биты первого операнда. При этом бит, уходящий за пределы ячейки, заносится в флаг CF, а с другого конца в операнд добавляется 0.

Логический сдвиг влево (shift left): `SHL op,1`
Логический сдвиг вправо (shift right): `SHR op,1`

Условно действия этих команд можно изобразить так (слева – для SHL, справа – для SHR):

Примеры:

```
MOV AL,01000111b
SHL AL,1           ;CF=0, AL=10001110b
MOV AL,01000111b
SHR AL,1           ;AL=00100011b, CF=1
MOV DH,00111000b
MOV CL,3
SHL DH,CL          ;CF=1, DH=11000000b
```

Напомним, что в памяти ПК величины размером в слово хранятся в "перевернутом" виде. Это учитывается командами сдвига, которые сдвигают слово так, как если бы оно было записано в неперевернутом виде. Например:

```
X DW 1Fh ;0000 0001 1111 1111b (X: 11111111b, X+1: 00000001b)
...
SHL X,1  ;0000 0011 1111 1110b (X: 11111110b, X+1: 00000011b)
```

Быстрое умножение и деление на степени 2

Одно из основных применений команд логического сдвига – это быстрое умножение и деление целых чисел на степени двойки, на 2^k. Конечно, эти операции можно выполнить с помощью команд умножения и деления, однако сдвигами эти операции выполняются значительно быстрее. Поэтому, если заранее известно, что число надо умножить или разделить на степень двойки, то лучше воспользоваться командами логического сдвига.

Что такое сдвиг десятичного числа, скажем 175, на 3 цифры влево? Это приписывание трех нулей справа, т. е. умножение на 10^3: 175000. Аналогично, сдвиг двоичного числа на k разрядов влево – это приписывание справа k двоичных нулей, т. е. умножение на 2^k. Например, при сдвиге числа 5 на 3 разряда влево получаем:

```
5=101b ---> 101000b = 40 = 5*2^3
```

Отметим, что это же верно и для отрицательных чисел, представленных в дополнительном коде. Например, при сдвиге числа -4 на 1 разряд влево получаем:

```
доп(-4) = 11111100b ---> 11111000b = 100h-8 = доп(-8)
```

В общем случае умножение на 2^k реализуется так:

```
;op:=op*2^k (op – число со знаком и без)
MOV CL,k
SHL op,CL
```

Однако надо учитывать, что все это верно, только если результат сдвига умещается в ячейку (это замечание носит общий характер: в любой операции результат будет правильным, только если он не "выходит" за размеры ячейки). Например, путем сдвига можно реализовать умножение на 2 всех беззнаковых чисел, меньших 128, и всех знаковых чисел от -64 до +63.

Что касается сдвига на k разрядов вправо, то это отбрасывание последних k разрядов, что соответствует получению неполного частного (операции div) от деления на 2^k. Например, при сдвиге числа 18 на 3 разряда вправо получаем:

```
18=10010b ---> 10b = 2 = 18 div 2^3
```

Однако так можно реализовать операцию деления нацело только для беззнаковых чисел, для отрицательных чисел это не проходит. Поэтому имеем:

```
;op:=op div 2^k (op – число без знака)
MOV CL,k
SHR op,CL
```

Как можно быстро получить остаток от деления на степень 2, будет рассмотрено в конце разд. 6.3.

6.2.2. Арифметические сдвиги

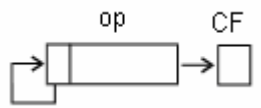
Команды арифметического сдвига предназначены для реализации быстрого умножения и деления знаковых чисел на степени двойки.

Арифметический сдвиг влево (shift arithmetic left): SAL op1,op2

Эта команда в точности совпадает с командой SHL (точнее, SAL и SHL – разные мнемонические названия одной и той же машинной команды), поскольку, как уже сказано, командой SHL можно реализовать умножение на степени двойки и чисел со знаком, и чисел без знака.

Арифметический сдвиг вправо (shift arithmetic rigth): SAR op1,op2

Как и в команде логического сдвига вправо, здесь также сдвигаются вправо все биты первого операнда, причем "уходящий" бит заносится в флаг CF, однако затем знаковый (самый левый) бит операнда восстанавливает свое исходное значение. Условно действие этой команды можно изобразить так:



Примеры:

```
MOV BH,10001110b
SAR BH,1        ;BH=11000111b, CF=0
MOV BH,00001110b
SAR BH,1        ;BH=00000111b, CF=0
```

Команда SAR может быть использована для быстрого получения неполного частного при делении знаковых чисел на степени 2. Однако эта операция отличается от той, что реализуется командой IDIV: если IDIV округляет частное в сторону 0, то SAR округляет в сторону минус бесконечности. Например, при делении -1 на 2 (т. е. при частном -0.5) команда IDIV выдаст ответ 0, а команда SAR – ответ -1:

```
MOV AL,-1  ;AL=11111111b
SAR AL,1   ;AL=11111111b (-1)
```

Что же касается деления неотрицательных чисел на степени 2, то команда SAR дает тот же ответ, что и команды SHR, DIV и IDIV.

6.2.3. Циклические сдвиги

Особенность циклических сдвигов в том, что "уходящий" бит не теряется, а возвращается в операнд, но с другого конца.

Циклический сдвиг влево (rotate left): `ROL op,1`
Циклический сдвиг вправо (rotate right): `ROR op,1`

В команде ROL все биты сдвигаются влево, причем самый левый бит возвращается в операнд с правого конца и одновременно заносится в флаг CF (см. слева), а в команде ROR все аналогично, только сдвиг выполняется вправо (см. справа):

Примеры:

```
MOV CL,11000011b
ROL CL,1   ;CF=1, CL=10000111b
MOV BH,11100010b
ROR BH,1   ;BH=01110001b, CF=0
```

Команды циклического сдвига обычно используются для перестановки частей содержимого ячейки или регистра. Например, поменять местами правую и левую половины регистра AL можно циклическим сдвигом этого байта на 4 разряда влево (или вправо):

```
MOV AL,17h  ;AL=00010111b
MOV CL,4
ROL AL,CL   ;AL=01110001b=71h
```

В ПК имеется еще две команды циклического сдвига:

Циклический сдвиг влево через перенос (rotate lefh through carry):

```
RCL op,1
```

Циклический сдвиг вправо через перенос (rotate rigth through carry):

```
RCR op,1
```

По команде RCL все биты первого операнда сдвигаются на одну позицию влево, причем самый левый бит попадает в флаг CF, а прежнее значение этого флага заносится в самый правый разряд операнда (см. слева); в команде RCR все аналогично, только осуществляется сдвиг вправо (см. справа):

Примеры:

```
MOV BL,11110000b ;пусть CF=0
RCL BL,1         ;CF=1, BL=11100000b
RCL BL,1         ;CF=1, BL=11000001b
```

Эти команды обычно используются при переносе битов из одного регистра (или переменной) в другой. Например, сдвинуть на 3 разряда влево значения регистров AL и DH, приписав справа к AL три левых бита регистра DH, можно так:

```
   MOV CX,3
L: SHL DH,1
   RCL AL,1
   LOOP L
```

6.2.4. Команды сдвига в процессорах 80186 и старше

Мы рассмотрели команды сдвига процессора 8086. Они не очень удобны в том смысле, что величину сдвига, если она больше 1, надо указывать через регистр CL, даже если она известна заранее. Поэтому, например, сдвиг значения регистра AX на 3 разряда влево приходится реализовывать либо трехкратным сдвигом на 1 разряд, либо с помощью регистра CL, который тем самым портится:

```
SHL AX,1   или  MOV CL,3
SHL AX,1   SHL AX,CL
SHL AX,1
```

В процессоре 80186 были введены новые команды сдвига, в которых в качестве второго операнда можно указывать любую величину сдвига, а не только 1. В ЯА эти команды записываются так:

```
SHL op,i8  SHR op,i8  ROL op,i8        и т. д.
```

Здесь второй операнд является непосредственным, имеет размер байта и трактуется как беззнаковое число. Например, сдвиг значения AX на 3 разряда влево реализуется командой

```
SHL AX,3
```

Однако при использовании этих команд надо учитывать следующее. Как уже отмечалось в разд. 3.4, по умолчанию ассемблер разрешает указывать в программах на ЯА только команды процессора 8086 и "ругается", если в тексте оказались команды, появившиеся в последующих моделях процессоров семейства 80x86. Так вот, чтобы можно было воспользоваться новыми командами сдвига, надо в любом месте текста программы, но до первой такой команды, поместить директиву .186:

```
.186
...
SHL AX,3
...
```

Напомним, что в данной книге мы ограничиваем себя только командами процессора 8086, базового в семействе 80x86, поэтому в дальнейшем мы не будем пользоваться рассмотренными здесь вариантами команд сдвига.

6.3. Упакованные данные

Одна из основных причин, по которым программу пишут на ЯА, а не на языке высокого уровня, – это когда надо получить программу, экономно расходующую память. А один из главных способов экономии памяти является использование упакованных данных, когда в ячейках памяти размещают не по одной величине, а по несколько величин, когда размещение данных ведется не на уровне байтов и слов, а на уровне отдельных битов.

Для примера рассмотрим величины типа DATE (дата), включающие в себя поля Y (год), M (номер месяца) и D (день), причем под "годом" будем понимать две последние десятичные цифры года. При этом условии на каждое поле можно отвести байт. Однако такое представление не очень экономно. В самом деле, две последние цифры года – это величина, меньшая $128=2^7$, поэтому на поле Y можно отвести 7 бит, а мы отводим 8 бит и теряем на этом 1 бит. Номер месяца меньше $16=2^4$, поэтому для поля M достаточно 4 бит, а мы отводим 8 и теряем на этом 4 бита. День меньше $32=2^5$, поэтому для него достаточно 5 бит, а не 8, поэтому мы теряем 3 бита. Итого, мы теряем 8 бит, т. е. целый байт, на одной дате. Конечно, это мелочь, но только если в программе используется всего несколько дат. Если же в программе используются сотни, тысячи дат, тогда потери уже станут ощутимыми. В такой ситуации надо уже по-иному, более экономно представлять данные. Надо под каждое

поле отводить не байт, а только часть байта – столько битов, сколько действительно надо и не больше, упаковывая в одной ячейке сразу несколько полей. В нашем случае под одну дату достаточно отвести слово, 16 бит, распределив их, например, так:

← размеры полей в битах

При таком представлении мы сэкономим по байту на каждой дате.

Подобного рода данные и называются упакованными.

Какие проблемы возникают при работе с упакованными данными? Вспомним, что в любой ЭВМ содержимое ячейки, т. е. машинное слово, можно считывать или записывать только целиком, но не частями. Поэтому при работе с упакованными данными возникают две основные проблемы:

- как выделять часть содержимого ячейки,
- как составлять содержимое ячейки из отдельных частей.

Решение этих проблем мы сейчас и рассмотрим.

Выделение части машинного слова

Пусть, к примеру, байт X разделен на следующие части:

← размеры полей в битах

и нам надо проверить, равна ли часть B пяти (101b). Ясно, что сравнивать этот байт с байтом 00101000b нельзя, т. к. будут мешать части A и C. Нельзя сравнивать и только часть битов ячейки, не сравнивая другие биты, – нет таких команд. Что делать?

В подобных ситуациях применяется следующий искусственный прием: в байте сохраняют только те биты, которые составляют интересующую часть, а остальные биты обнуляют, и затем с полученным таким образом машинным словом обращаются так, как если бы это была только часть ячейки. В нашем случае надо байт X преобразовать в байт 00 B 000 и затем сравнение поля B с числом 101b заменить на сравнение этого байта с байтом 00101000b. Ясно, что последние два байта будут равны тогда и только тогда, когда B=5. Здесь нам уже не будут мешать остальные части исходного байта, ради этого мы их и обнуляли.

Оставление только нужных битов ячейки и обнуление других битов называется выделением части ячейки. Как оно реализуется? Для этого используется команда логического умножения AND, что объясняется следующим свойством конъюнкции:

```
для любого X: 1 and X = X, 0 and X = 0,
```

т. е. 1 в одном операнде сохраняет соответствующий бит в другом операнде, а 0 очищает этот бит. С учетом этого для выделения части ячейки надо ее со-

держимое логически умножить на машинное слово, содержащее единицы в нужных разрядах и нули в остальных разрядах:

	A	B	C		
and	00	111	000	←	маска
	00	B	000		

Отметим, что машинное слово, которое используется для выделения какой-то части ячейки, принято называть маской.

С учетом всего сказанного, наша задача сравнения поля B из байта X с чис-лом 5 решается так:

```
   MOV AL,X           ;AL: A B C
   AND AL,00111000b   ;AL: 0 B 0
   CMP AL,00101000b   ;AL = 0 5 0 ? (B=5?)
   JE YES             ;B=5 -> YES
NO: ...
```

Составление машинного слова из отдельных частей

Теперь рассмотрим операцию, обратную к выделению части машинного слова, а именно операцию составления машинного слова из отдельных частей.

Пусть, к примеру, имеются следующие байты X1 и X2 и из них надо составить такой байт X:

X1	A	0	C
X2	0	B	0
X	A	B	C

Оказывается, что в данном случае полезна команда логического сложения OR. Это объясняется следующим свойством дизъюнкции:

```
для любого X: 0 or X = X
```

т. е. 0 в одном операнде сохраняет соответствующий бит другого операнда. Поэтому наша задача решается так:

```
MOV AL,X1
OR AL,X2
MOV X,AL
```

Отметим, однако, что такой способ объединения частей проходит, только если биты нужной части логически складываются с нулями в тех же позициях другого байта. Если же там не нули, то так соединять части нельзя, поскольку при дизъюнкции единички "забивают" биты в другом операнде:

```
для любого X: 1 or X = 1
```

Например, если бы байт X1 имел вид A 111 C, то при логическом сложении его с байтом X2 получился бы байт A 111 C, а не A B C.

Итак, мы рассмотрели, как можно выделять части машинных слов и как можно составлять слова из отдельных частей. На основе этих операций реализуются уже более сложные действия над частями машинных слов. Например, преобразование байта X: A B C --> A B B, что соответствует замене части C на B, можно реализовать по следующей схеме:

- обнуляем в X часть C, т. е. выделяем части A и B:
```
AND X,11111000b   ; X: A B 0
```
- выделяем часть B:
```
MOV AL,X
AND AL,00111000b  ;AL: 0 B 0
```
- сдвигаем B вправо:
```
MOV CL,3
SHR AL,CL         ;AL: 0 0 B
```
- соединяем все части вместе:
```
OR X,AL           ; X: A B B
```

С помощью подобных выделений, сдвигов и объединений частей машинных слов реализуются и любые другие операции над упакованными данными. Как видно, здесь активно используются логические команды и команды сдвига. Именно обработка упакованных данных и является основной областью применения этих команд.

И еще одно замечание. Уже из нашего последнего примера видно, что обработка упакованных данных – вещь более сложная, чем обработка неупакованных данных, и требует больше времени. Поэтому, когда решается вопрос о том, упаковывать данные или нет, то прежде всего надо определить, что для нас важнее – экономия памяти или экономия времени.

В заключение рассмотрим еще одно применение логических команд. Как уже было сказано, с помощью команд сдвига можно быстро находить неполное частное при делении на степени двойки. Оказывается, для беззнаковых чисел можно быстро получать и остаток от такого деления, т. е. быстро выполнять операцию op:=op mod 2^k. Для этого надо всего лишь выделить в делимом op правые k бит (аналогия в десятичной системе: остаток от деления op на 1000 – это три правые цифры числа op). Например:

```
AND AX,111b  ;AX:=AX mod 2^3
```

6.4. Множества

В гл. 5 были рассмотрены такие составные типы данных, как массивы и структуры. Теперь рассмотрим еще один составной тип данных – множества языка Паскаль. Это будет и хорошим примером на обработку упакованных данных.

Практически ни в одной ЭВМ нет какого-либо стандартного представления для множеств, нет и средств для работы с ними. Поэтому программистам приходится самим выбирать представление для множеств, самим реализовы-

вать операции над ними. Мы рассмотрим один из возможных вариантов того, как это делается.

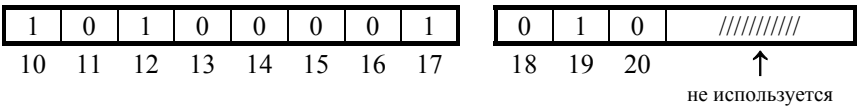
6.4.1. Машинное представление множеств

Пусть имеется множество M из целых чисел, принадлежащих отрезку [l, r], где l и r – целые числа (l<=r):

```
var M: set of l..r;
```

Отведем для этого множества несколько соседних байтов памяти и поставим в соответствие каждому числу из отрезка [l,r] один бит одного из этих байтов по следующему правилу: числу l – самый левый бит самого первого байта, числу l+1 – следующий бит этого байта и так бит за битом сначала в первом байте, затем в следующем байте и т. д. Заполняем же эти байты следующим образом: если число принадлежит множеству, то в соответствующий ему бит записываем 1, не принадлежит – записываем 0.

Например, при l=10 и r=20 множество M=[10,12,17,19] имеет следующее представление:



Проведем кое-какие подсчеты.

Нетрудно подсчитать, что число байтов, которые надо отвести под множество, равно (r-l) div 8+1. Например, при l=10 и r=20 эта величина равна (20–10) div 8+1= =1+1=2. В связи с этим в программе на ЯА переменную M надо описывать так:

```
M DB (r-l)/8+1 DUP(0)
```

Отметим, что при нулевых начальных значениях байтов это описание соответствует пустому множеству.

Теперь определим, какой бит какого байта соответствует числу k из отрезка [l, r]. Если вести отсчет от начального байта, выделенного для множества, и считать его нулевым и если биты в каждом байте нумеровать слева направо начиная с 0, тогда получаем такие формулы:

```
номер байта = (k-l) div 8
номер бита = (k-1) mod 8
```

Конкретные примеры при l=10:

```
k=12: номер байта = (12-10) div 8 = 0
      номер бита = (12-10) mod 8 = 2

k=18: номер байта = (18-10) div 8 = 1
      номер бита = (18-10) mod 8 = 0
```

6.4.2. Реализация операций над множествами

Теперь рассмотрим, как реализуются операции над множествами при таком их представлении. Мы опишем две операции – проверку на принадлеж-

ность элемента множеству и объединение множеств. Остальные операции над множествами реализуются аналогично.

Проверка на принадлежность элемента множеству

Пусть имена L и R описаны в программе как константы (L<=R) и пусть множество M и переменная K:

```
M DB (R-L)/8+1 DUP (?)  ;set of L..R
K DW ?                  ;L<=K<=R
```

получили в программе какие-то значения и надо проверить, принадлежит ли число K множеству M.

Согласно нашему представлению множеств, если число входит в множество, то в соответствующем бите находится 1, не входит – в этом бите находится 0. Следовательно, для проверки, входит ли число в множество, надо сравнить этот бит с 1. Чтобы сделать это, надо сначала выделить тот байт множества, которому принадлежит интересующий нас бит (как вычислить номер этого байта, мы уже знаем), а затем уже в этом байте надо выделить нужный бит (номер его мы также умеем вычислять). Все это можно реализовать так:

```
   MOV AX,K
   SUB AX,L       ;AX=K-L
   MOV BH,8
   DIV BH         ;AH=номер бита , AL=номер байта
   MOV BL,AL
   MOV BH,0       ;BX=номер байта (от начала M) как слово
   MOV AL,M[BX]   ;AL - сам байт
   MOV CL,AH      ;CL=номер нужного бита (отсчет слева)
   SHL AL,CL      ;сдвиг бита к левому краю AL
   TEST AL,80h    ;выделение только этого бита (ZF=1,
                  ;если бит=0)
   JZ NO          ;K не принадлежит M --> NO
YES:              ;K принадлежит M
```

Объединение множеств

Пусть имеются множества M, M1 и M2, элементами которых являются числа из отрезка [l, r], и надо реализовать операцию объединения множеств M1 и M2 с записью множества-результата в M (в обозначениях языка Паскаля: M:=M1+M2).

Ясно, что каждый бит множества M зависит только от соответствующих битов в M1 и M2, причем правило зависимости такое: этот бит равняется 1 (соответствующее число принадлежит M), если есть 1 в соответствующем бите хотя бы одного из множеств M1 и M2 (если данное число принадлежит хотя бы одному из этих множеств). Нетрудно понять, что эта зависимость соответствует логическому сложению. При этом, конечно, данную операцию надо применить одновременно ко всем битам всех бай-

тов, занимаемых множествами, т. е. надо в цикле логически сложить все байты множеств M1 и M2.

```
   MOV CX,(R-L)/8+1  ;число байтов в множествах
   MOV BX,0          ;номер байта от начала множества
L: MOV AL,M1[BX]
   OR AL,M2[BX]      ;"объединение" байтов
   MOV M[BX],AL
   INC BX
   LOOP L
```

Аналогично реализуются и остальные теоретико-множественные операции (пересечение, разность и т. д.), надо лишь воспользоваться другими логическими операциями.

6.5. Записи

Как видно из предыдущего рассказа, работать с упакованными данными можно с помощью только тех средств, которые мы уже знаем. Однако ЯА предоставляет дополнительные средства, делающие эту работу более удобной. Этим средством являются записи, которые и будут рассмотрены в этом разделе.

Запись – это упакованная структура, когда поля занимают не отдельные ячейки, а части ячеек. Более точно, в ЯА записью называется ячейка размером в байт или слово (другие размеры не допускаются), содержимое которой рассматривается как состоящее из нескольких частей, называемых полями записи. Каждое поле – это группа соседних битов. Поля должны быть прижатыми друг к другу, между ними не должно быть промежутков, не относящихся к полям. Размер поля (число битов в нем) любой, но в сумме их размеры не должны превышать 16. Сумма размеров всех полей записи называется размером записи. Если размер записи меньше или равен 8, то под запись ассемблер будет отводить байт памяти, при размере от 9 до 16 – будет отводить слово. Если поля занимают не всю ячейку, то они прижимаются к правому краю ячейки, а левые биты ячейки считаются не относящимися к записи. Поля получают имена, однако, как мы увидим, обращаться к полю записи по имени нельзя.

6.5.1. Описание типа записи

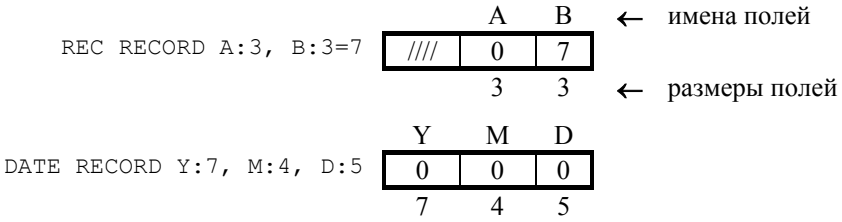
Прежде чем использовать в программе записи, надо описать их тип, что делается с помощью директивы следующего вида:

```
<имя типа записи> RECORD <поле> {, <поле>}
```

где <поле> ::= <имя поля> : <размер> [= <выражение>]

<размер> и <выражение> – константные выражения ("?" не допускается)

Примеры:



При описании типа записи все ее поля перечисляются слева направо. Для каждого поля обязательно указывается имя и (через двоеточие) размер в битах. Имена полей должны отличаться от всех других имен, используемых в программе. При описании поля может быть указаны знак = и значение этого поля по умолчанию (это значение должно "укладываться" в размер поля); если такое значение не указано, то значением поля по умолчанию считается 0. В этом отличие от структур, и связано оно с тем, что оставить часть ячейки (поле) ничем не заполненной попросту нельзя. По этой же причине нулевыми будут и левые биты ячейки, не относящиеся к записи.

Описание типа записи носит чисто информационный характер, по нему ассемблер ничего не заносит в машинную программу, поэтому его можно размещать в любом месте программы, но обязательно до описания переменных этого типа.

6.5.2. Описание переменных-записей

Переменные-записи называются просто записями. Для описания такой переменной в программу надо поместить директиву следующего вида:

```
имя_переменной имя_типа <нач_знач {, нач_знач}>
```

где нач_знач – это константное выражение, ? или "пусто" (здесь уголки – не метасимволы, а явно указываемые символы).

Примеры:

```
 R1 REC <5,>       ;R1: 0 5 7
 R2 REC <,?>       ;R2: 0 0 0
VIC DATE <45,5,9>  ;VIC: 45 5 9
```

Смысл подобной директивы такой же, как и директивы, описывающей переменную-структуру: по ней отводится в памяти место (байт или слово) под запись, которой дается имя, а ее полям – начальные значения. Отличие проявляется лишь в том, что знак "?" означает нулевое начальное значение. Если же указано "пусто", то в качестве начального значения берется значение по умолчанию, заданное при описании типа данной переменной.

Допускается и следующее сокращение при задании начальных значений полей: если в качестве последних нач_знач указывается "пусто", то последние подряд идущие запятые можно опустить.

Пример:

```
D1 DATE <97,,>  эквивалентно D1 DATE <97>
D2 DATE <,,>    эквивалентно D2 DATE <>
```

(уголки опускать нельзя).

Как и в случае структур, одной директивой можно описать массив записей, для чего в правой части директивы указывается несколько операндов и/или конструкция повторения DUP. Например, по директиве

```
X DATE 100 DUP(<>)
```

выделяется место сразу под 100 записей типа DATE, причем поля всех этих записей будут иметь одинаковые начальные значения – те, что указаны при описании этого типа.

6.5.3. Средства для работы с полями записей

Как видно, пока все очень похоже на структуры. Однако дальше идут отличия. Дело в том, что на поля записи нельзя уже ссылаться с помощью конструкции вида <имя записи>.<имя поля>; например, VIC.Y – ошибка. Это связано с тем, что такая конструкция должна обозначать не ячейку, а только часть ячейки, но части ячеек не адресуются. Поэтому такая конструкция и не допускается в ЯА. Работа с полями записей ведется иначе, она не так удобна, как работа с полями структур, но все-таки определенные удобства имеются. Рассмотрим их.

Прежде всего отметим, что при работе с записями как едиными объектами никаких проблем не возникает, т. к. запись – это байт или слово, а уж с байтами и словами мы работать умеем. Например, присваивание R1:=R2 реализуется командами:

```
MOV Al,R2
MOV R1,AL
```

Поэтому поговорим о средствах ЯА для работы с полями записей.

Оператор WIDTH: *WIDTH <имя поля записи>*
WIDTH <имя записи или ее типа>

Если указано имя поля, то значением оператора является размер указанного поля в битах, а если указано имя записи или имя типа записи, тогда оператор выдает размер всей записи (сумму размеров всех ее полей). Например:

```
WIDTH Y = 7
WIDTH REC = 6
```

Оператор MASK: *MASK <имя поля записи>*
MASK <имя записи или ее типа>

Значением оператора является маска (байт или слово – в зависимости от размера записи), содержащая 1 в тех разрядах, которые принадлежат указанному полю или всем полям записи, и нули в остальных разрядах. Например:

```
MASK A   = 00111000b
MASK B   = 00000111b
MASK REC = 00111111b
MASK M   = 0000000111100000b
```

Оператор MASK используется при выделении полей записей. Выгода от него в том, что не надо явно выписывать 1 и 0 для маски, за нас это сделает ассемблер. Например, проверить, содержит ли поле D записи VIC число 9, можно так:

```
    MOV AX,VIC
    AND AX,MASK D
    CMP AX,9
    JE YES
NO:
```

Запись MASK D более наглядна, чем запись 11111b или 1Fh.

Значение имени поля

Имени любого поля записи ассемблер приписывает некоторое значение. Им является число, на которое надо сдвинуть это поле вправо, чтобы оно оказалось прижатым к правому краю ячейки. Например, значением имени D (из записи типа DATE) является число 0, имени M – число 5, имени Y – число 9.

Встречая имя поля записи в тексте программы, ассемблер заменяет его на это значение. Обычно это нужно в командах сдвига, в которых мы сдвигаем поля записей. Например, проверить, равно ли пяти поле M в записи VIC, можно так:

```
    MOV AX,VIC     ;AX: Y M D
    AND AX,MASK M  ;AX: 0 M 0
    MOV CL,M
    SHR AX,CL      ;AX: 0 0 M
    CMP AX,5       ;M=5?
    JE YES
NO:
```

Таковы средства, предоставляемые в ЯА для работы с записями. Они, конечно, хуже средств для работы со структурами, но это и понятно: в самом ПК нет удобных команд для работы с частями ячеек, и это отражается в ЯА.

7. ПРОГРАММНЫЕ СЕГМЕНТЫ

До сих пор мы считали, что в ПК используются 16-разрядные адреса. Но пора вспомнить, что объем памяти ПК равен 2^{20} байт (1 Мб), а это требует 20-разрядных адресов. В данной главе рассматривается, как наличие таких 20-разрядных адресов сказывается на программах и как выглядит полная программа на ЯА.

7.1. Сегментирование адресов в ПК

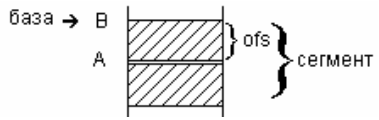
7.1.1. Общая схема базирования адресов

ПК относится к ЭВМ, в которых ради сокращения размера команд используется базирование адресов. Поэтому прежде всего рассмотрим, что такое базирование адресов в ЭВМ.

Если в оперативной памяти ЭВМ имеется 2^k ячеек, то для ссылок на эти ячейки нужны k-разрядные адреса; будем называть их абсолютными адресами (другое название – физические адреса). Ясно, что при большом объеме памяти большим будет и размер абсолютных адресов, а это ведет к увеличению длины команд, к увеличению размера программ в целом. Это плохо. Чтобы сократить размеры команд, поступают следующим образом.

Память условно делят на участки, которые принято называть сегментами. Начальные адреса сегментов (эти адреса называются базами) могут быть любыми, но на длины сегментов накладывается ограничение: размер любого сегмента не должен превосходить, скажем, 2^m ячеек, где m<k.

В этих условиях абсолютный адрес A любой ячейки памяти можно представить в виде суммы A=B+ofs, где B – база сегмента, к которому относится ячейка A, а ofs – смещение (offset) или относительный адрес ячейки, т. е. ее адрес, отсчитанный от начала сегмента, от адреса B:



Ограничение размера сегментов означает, что 0<=ofs<=2^m-1, и потому для записи смещений достаточно m разрядов. Следовательно, в сумме A=B+ofs большая часть адреса A приходится на базу B, а ofs – это лишь небольшой добавок. Учитывая это, поступаем следующим образом. Если в команде надо указать абсолютный адрес A, то большее слагаемое – базу B – "упрятываем" в некоторый регистр R, а в команде указываем лишь этот регистр и меньшее слагаемое ofs, т. е. вместо команды

```
КОП ... A ...
```

используем команду

```
КОП ... R ofs ...
```

Что это дает? Поскольку команда работает с исполнительным адресом, а он вычисляется по формуле [R]+ofs=B+ofs=A, то эта команда будет работать с нужным нам адресом A. С другой стороны, поля R и ofs занимают мало места в команде, поэтому размер команды уменьшится – по сравнению с тем случаем, когда в команде указывается абсолютный адрес. К этому мы и стремились.

Отметим попутно, что, записав один раз в регистр R базу сегмента, далее мы можем, не меняя значение этого регистра, использовать его для доступа ко всем ячейкам данного сегмента. И если данные программы располагаются в памяти компактно, то засылок баз в регистры будет немного.

Регистры, в которых хранятся базы, принято называть базовыми регистрами, а способ записи в командах не абсолютных адресов, а базовых регистров и смещений, называется базированием адресов.

Такова общая схема базирования адресов, используемая во многих ЭВМ. В ПК используется эта же схема, но с рядом особенностей, которые мы сейчас и рассмотрим.

7.1.2. Особенности сегментирования адресов в ПК

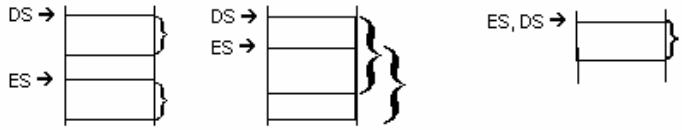
Прежде всего отметим, что в ПК вместо термина "базирование адресов" используется термин "сегментирование адресов", а базовые регистры называют сегментными регистрами. Кроме того, размер памяти в ПК равен 1 Мб, т. е. 2^{20} байт (k=20), поэтому абсолютные адреса здесь 20-разрядные, а размеры сегментов не должны превышать величину 64 Кб, т. е. 2^{16} байт (m=16), поэтому смещения здесь – это 16-разрядные адреса.

Первая особенность сегментирования адресов в ПК заключается в том, что в качестве сегментных регистров можно использовать не любые регистры, а лишь регистры CS, DS, SS и ES. Конечно, начальные адреса сегментов памяти можно хранить и в других регистрах (скажем, в AX), однако использовать эти другие регистры для сегментирования адресов не удастся.

Поскольку в ПК всего четыре сегментных регистра, то одновременно можно работать с четырьмя сегментами памяти: устанавливаем на начало каждого из этих сегментов свой регистр и далее сегментируем по нему все адреса из этого сегмента. А что делать, если надо работать с большим числом сегментов памяти, скажем, с пятью? Когда потребуется доступ к ячейкам из пятого сегмента, выбираем один из сегментных регистров и где-то спасаем (в обычном регистре или ячейке памяти) его содержимое, а затем записываем в этот регистр начало пятого сегмента и далее используем этот регистр для доступа к ячейкам пятого сегмента. Позже, если надо, можно восстановить прежнее значение этого регистра, чтобы он снова указывал на один из пер-

вых 4 сегментов. Меняя таким образом значения сегментных регистров, можно работать с любым числом сегментов памяти.

Отметим, что взаимное расположение сегментов памяти может быть любым. Они могут не пересекаться (см. слева), могут пересекаться (см. в середине), а могут и полностью совпадать (см. справа), т. е. два сегментных регистра могут указывать на начало одного и того же сегмента:



(Ячейки из пересечения могут сегментироваться сразу по нескольким сегментным регистрам.) Какие именно сегменты памяти использовать – это личное дело автора программы.

Размеры сегментов также определяются автором программы, лишь бы эти размеры не превосходили 64 Кб. Установив сегментный регистр на начало сегмента, мы потенциально можем по нему просегментировать 64 Кб, ну а сколько из них реально будет использовано, т. е. каков размер сегмента, уже определяет автор программы.

Теперь о второй особенности сегментирования адресов в ПК. При записи команд на ЯА ссылка на сегментный регистр указывается с помощью следующей конструкции:

```
<сегментный регистр>:<адресное выражение>
```

которая называется адресной парой и которая "говорит", что адрес, являющийся значением выражения, должен быть просегментирован по указанному регистру. Например, в команде MOV AX,ES:X адрес переменной X будет сегментироваться по регистру ES.

Запись CS:, DS:, SS: или ES: принято называть префиксом сегментного регистра или просто префиксом (приставкой). В ЯА префикс всегда записывается перед адресом, который должен быть просегментирован. Однако в машинном языке ситуация несколько иная: здесь префикс ставится перед всей командой. Например, команда MOV AX,ES:X на самом деле записывается в виде двух машинных команд:

```
ES:
MOV AX,X
```

ES: – это специальная команда (без операндов), которая и называется префиксом. Всего таких команд-префиксов четыре, по одной на каждый сегментный регистр. Сама по себе такая команда-префикс ничего не делает, но она влияет на следующую за ней команду: префикс "говорит", что адресный операнд в следующей команде должен быть просегментирован по соответствующему регистру (в нашем примере – по регистру ES). Если префикс по-

ставлен перед командой, где нет адресного операнда, то он проработает как "пустая" команда.

Итак, в машинных командах сегментные регистры указываются не внутри команд, а перед ними (в виде команд-префиксов). Зачем это сделано, будет объяснено чуть позже. А пока отметим, что в ЯА записывать префикс вне команды нельзя; например, запись ES: MOV AX,X считается ошибочной с точки зрения ЯА. Префикс обязательно должен указываться внутри команды и непосредственно перед адресом.

Третья особенность сегментирования адресов в ПК связана с размером сегментных регистров. Общая схема базирования адресов предполагает, что размеры базовых регистров достаточно большие и в них может быть размещен любой абсолютный адрес, любая база. Однако в ПК это условие не выполняется: в ПК абсолютные адреса – 20-разрядные, а все регистры, в том числе и сегментные, 16-разрядные. Естественно, возникает вопрос: как в 16-разрядных сегментных регистрах удается разместить 20-разрядные базовые адреса?

В ПК эта проблема решена следующим образом. Мы до сих пор считали, что в качестве базы можно использовать любой адрес. В ПК же на начальные адреса сегментов накладывается ограничение: в качестве базы можно использовать любой адрес, но кратный 16. Особенность этих адресов в том, что у них последние 4 бита нулевые, или, что то же самое, в шестнадцатеричной записи этих адресов последняя цифра всегда нулевая, т. е. они имеют вид XXXX0h, где X – любая цифра. А раз так, то эту нулевую цифру можно явно не указывать, а лишь подразумевать. Так и делают: в сегментных регистрах хранят только первые 16 бит начального адреса сегмента, т. е. первые четыре шестнадцатеричные цифры. Например, если началом сегмента является адрес 12340h, то в сегментном регистре будет храниться величина 1234h. Начальный адрес сегмента без последнего шестнадцатеричного 0 называют номером сегмента и обозначают как seg.

Центральный процессор, естественно, учитывает эту особенность сегментных регистров и при сегментировании адреса прежде всего к содержимому сегментного регистра приписывает справа этот неуказанный 0 и только затем прибавляет смещение, указанное в команде. Условно это можно изобразить так:

```
CP=XXXX                    XXXX0 – база
КОП ... CP:YYYY ... ==> + _YYYY – смещение
                           ZZZZZ – абсолютный адрес
```

Поскольку приписывание справа нуля к шестнадцатеричному числу эквивалентно умножению этого числа на 16, то формулу вычисления абсолютного адреса по адресной паре CP:ofs можно выразить так:

```
Аабс = 16*[CP] + ofs
```

Например, если ES=1234h, тогда адресная пара ES:53h задает абсолютный адрес 16*1234h+53h = 12340h+53h = 12387h.

Теперь уточним одну вещь. Как мы знаем, адреса, указываемые в командах, можно модифицировать по регистрам BX, BP, SI и DI. Как сочетается эта модификация с сегментированием? Правила здесь следующие. Сначала выполняется модификация адреса по регистрам-модификаторам, в результате чего получается адрес, который мы называем исполнительным. При этом, напомним, вычисление ведется по модулю 2^{16}, т. е. исполнительный адрес – это всегда 16-разрядный адрес. Затем этот адрес рассматривается как смещение и именно он сегментируется, т. е. именно к нему добавляется содержимое сегментного регистра, умноженное на 16. Причем данное суммирование ведется по модулю 2^{20}, чтобы не получился адрес, больший максимально допустимого. Таким образом, более точная формула вычисления абсолютного адреса такова:

$$Аабс = (Аисп + 16*[СР]) \bmod 2^{20}$$

Отметим, что сегментирование исполнительного адреса происходит, только если команда осуществляет доступ к памяти. Если же в команде не предусмотрено обращение к памяти, то сегментный регистр не учитывается. Скажем, команда LEA r16,A не обращается к памяти, поэтому адрес A не будет сегментироваться, т. е. по этой команде выполняется операция r16:=Аисп, а не операция r16:=Аабс.

И, наконец, рассмотрим еще одну, наиболее интересную особенность сегментирования адресов в ПК.

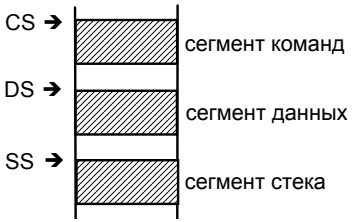
7.1.3. Сегментные регистры по умолчанию

Если проанализировать реальные машинные программы, то можно заметить, что в них в общем-то указываются адреса всего из трех областей памяти – сегмента команд, сегмента данных и сегмента стека. Например, в командах перехода всегда указываются адреса других команд, т. е. это ссылки на сегмент команд – ту область памяти, где располагаются команды программы. В командах, работающих со стеком, естественно, указываются адреса из сегмента стека – области памяти, занимаемой стеком. В остальных же командах (пересылках, арифметических и т. п.) указываются, как правило, адреса из сегмента данных – той области памяти, где расположены переменные, массивы и другие данные программы.

С учетом этой особенности реальных программ в ПК принят ряд соглашений, которые позволяют во многих командах не указывать явно сегментные регистры, а подразумевать их по умолчанию.

Во-первых, договариваются, чтобы начальные адреса этих трех сегментов памяти всегда находились в определенных сегментных регистрах, а именно:

регистр CS должен указывать на начало сегмента команд, регистр DS – на начало сегмента данных, а регистр SS – на начало сегмента стека:



Что же касается регистра ES, то относительно него нет никаких договоренностей, он свободен и может указывать на любой сегмент памяти.

Во-вторых, договариваются о следующем. Перед адресами перехода регистр CS явно не указывается, он подразумевается по умолчанию; например, в команде перехода достаточно указать только метку (JMP L), префикс же CS: будет добавлен автоматически (JMP CS:L). При ссылках на сегмент данных можно явно не указывать регистр DS, он подразумевается по умолчанию; например, если X – имя переменной, то вместо команды MOV DS:X,0 можно писать просто MOV X,0. И, наконец, договариваются, что при ссылках на стек можно явно не указывать регистр SS, он подразумевается по умолчанию.

Более точно правила о выборе сегментных регистров по умолчанию формулируются следующим образом:

- Адреса перехода всегда сегментируются по регистру CS.
- В так называемых строковых командах действуют особые соглашения (см. разд. 10.1).
- Во всех остальных командах:
  - если адрес в команде не модифицируется или если он модифицируется, но среди модификаторов нет регистра BP, то этот адрес считается ссылкой в сегмент данных и потому по умолчанию сегментируются по регистру DS;
  - если адрес модифицируется по BP, то он считается ссылкой в стек и потому по умолчанию сегментируется по регистру SS.

(Здесь следует уточнить одну вещь, связанную с косвенными переходами. Рассмотрим такой фрагмент:

```
   X DW L
   ...
   JMP X    ; JMP DS:X = goto CS:L
   ...
L: ...
```

В команде перехода X – это еще не адрес перехода, а адрес той ячейки, где находится адрес перехода. Поэтому в данной команде подразумевается, что X является адресом из сегмента данных и потому по умолчанию сегмен-

тируется по регистру DS. Но вот когда из этой ячейки выбран адрес перехода (L), то уж он-то сегментируется по регистру CS.)

Итак, если в команде не указан сегментный регистр, то он выбирается согласно этим правилам. Но если сегментный регистр указан явно, тогда эти правила не действуют, а используется указанный регистр.

Что дают рассмотренные соглашения о сегментных регистрах? А то, что если их придерживаться, тогда в большинстве команд не надо явно указывать сегментные регистры, поскольку нужные в командах регистры как раз совпадают с регистрами, подразумеваемыми по умолчанию. Именно по этой причине все программы, которые мы до сих пор писали, были правильными, хотя мы в них никакие сегментные регистры и не указывали.

Выгода от этих соглашений также и в том, что они позволяют сократить размеры машинных программ, поскольку перед подавляющим большинством машинных команд можно не ставить команды-префиксы, команды проработают правильно и без них. Именно ради этого и пошли на "вынос" префиксов за пределы машинных команд: если бы префикс указывался в самой команде, то соглашения ничего не дали бы – место для префикса все равно оставалось бы в команде, и это место все равно надо было бы чем-то заполнять. Префиксы же как отдельные команды можно опускать, не указывать.

Итак, если придерживаться соглашений о сегментных регистрах, то текст программы на ЯА будет короче и размер машинной программы будет меньше.

А что будет, если этих соглашений не придерживаться? Тогда придется в командах выписывать полные адресные пары. Например, если мы установили на начало области данных не регистр DS, а, скажем, регистр SS, тогда для засылки 0 в ячейку X из этой области нужно писать команду MOV SS:X,0; писать же команду MOV X,0 уже нельзя, т. к. она воспримется как MOV DS:X,0 и потому зашлет 0 неизвестно куда. Ясно, что лучше придерживаться указанных соглашений, если мы не хотим все время выписывать префиксы.

Отметим, что одно из этих соглашений нельзя нарушать ни в коем случае: регистр CS всегда обязан указывать на начало сегмента команд. Это объясняется тем, что в ПК адрес очередной команды, которую надо выполнить, всегда указывается парой регистров CS и IP, где указатель команд IP содержит смещение этой команды, отсчитанное от начала сегмента команд. Поэтому, если регистр CS будет указывать не на начало сегмента команд, то поведение машины будет непредсказуемым. По этой же причине перед адресами перехода нельзя указывать префикс, отличный от CS: (префикс же CS: можно не указывать, поскольку он и так подразумевается по умолчанию).

Теперь рассмотрим такой вопрос: если придерживаться соглашений о сегментных регистрах, то существуют ли случаи, когда приходится явно указывать сегментные регистры? Да, такие случаи имеются, и их в общем-то два.

Первый – это когда мы в целом придерживаемся данных соглашений, но в каком-то частном случае хотим их нарушить. Пусть, к примеру, мы хотим записать в регистр AX содержимое ячейки L, расположенной в сегменте ко-

манд. Использовать здесь команду MOV AX,L нельзя, поскольку она воспримется как MOV AX,DS:L и потому запишет в AX содержимое ячейки, имеющей смещение L, но находящейся не в сегменте команд, а в сегменте данных. Как видно, нас здесь не устраивает соглашение о том, что в команде MOV по умолчанию берется регистр DS, поэтому мы обязаны явно указать тот сегментный регистр, который нам нужен, т. е. мы обязаны записать команду MOV AX,CS:L.

Второй случай – это когда нам надо работать с сегментом памяти, отличным от сегментов команд, данных и стека. В этом случае на начало этого сегмента устанавливают регистр ES и при ссылках на ячейки этого сегмента используют данный регистр, например: INC ES:Y. Опускать здесь префикс ES: нельзя, т. к. он не подразумевается по умолчанию. Отметим попутно, что регистр ES как раз и предназначен для работы с дополнительными сегментами памяти.

Итак, имеются случаи, когда приходится явно указывать сегментные регистры, но, к счастью, эти случаи встречаются редко, и в большинстве команд указываются не полные адресные пары, а только смещения.

7.2. Программные сегменты

Рассмотренные соглашения о сегментных регистрах заставляют нас следующим образом строить программу: в одном сегменте памяти надо разместить все команды программы и установить регистр CS на начало этого сегмента, в другом сегменте надо разместить все данные программы и на его начало установить регистр DS, а третий сегмент надо отвести под стек и установить на его начало регистр SS. Но предложения нашей программы на ЯА размещает в памяти ассемблер, а не мы. Как же ассемблер определяет, какие предложения в каком сегменте размещать? Ответ такой: он этого не знает, это должны сообщить ему мы сами. И делается это так: все предложения, которые мы хотим разместить в одном сегменте памяти, мы обязаны в тексте программы объединить в так называемый программный сегмент. Это аналог сегмента памяти, но в терминах ЯА. Программный сегмент имеет следующую структуру:

```
<имя сегмента> SEGMENT <параметры>
               <предложение>
                  ...
               <предложение>
<имя сегмента> ENDS
```

Программному сегменту дается имя, которое должно быть повторено дважды – в директиве SEGMENT, открывающей сегмент, и в директиве ENDS (end of segment), закрывающей его. Между этими директивами может быть указано любое число любых предложений. Смысл всей этой конструкции таков: мы заявляем ассемблеру, что все предложения между директивами SEGMENT и ENDS он должен разместить в одном сегменте памяти. Но при этом надо помнить, что размер сегмента памяти не должен превышать 64 Кб, поэтому указан-

ных предложений должно быть столько, чтобы в совокупности они занимали не более 64 Кб, иначе ассемблер зафиксирует ошибку.

Отметим, что в программе может быть несколько сегментов с одним и тем же именем. Считается, что это один сегмент, просто он по какой-то причине описан по частям; все предложения этих частей ассемблер соединит вместе.

Параметры директивы SEGMENT будут рассмотрены позже (в разд. 12.3), а пока лишь отметим, что параметры в основном нужны тогда, когда программа большая и хранится в нескольких файлах, и параметры указывают, как объединять эти файлы. Если же программа небольшая, умещается в одном файле, тогда параметры не нужны (кроме одного, о котором будет рассказано в этой главе).

Пример программных сегментов:

```
A SEGMENT
A1 DB 400h DUP(?)
A2 DW 8
A ENDS

B SEGMENT
B1 DW A2
B2 DD A2
B ENDS

C  SEGMENT
   ASSUME ES:A, DS:B, CS:C
L: MOV AX,A2
   MOV BX,B2
   ...
C  ENDS
```

Используя этот пример, уточним кое-какие вещи.

Выравнивание на границу параграфа

В общем случае ассемблер размещает в памяти программные сегменты последовательно друг за другом, причем каждый из них он обязательно размещает с адреса, кратного 16, – с границы так называемого параграфа (параграфом в ПК называют участок памяти, начинающийся с адреса, кратного 16, и состоящего из 16 байт). Пусть, например, сегмент A размещен с абсолютного адреса 10000h. Тогда все предложения этого сегмента займут в памяти байты с адресами от 10000h до 10401h. После этого первый свободный байт будет иметь адрес 10402h. Он не кратен 16 (не оканчивается на 0) и с него нельзя размещать сегмент B, поэтому ассемблер пропускает этот и последующие байты вплоть до ближайшего байта с адресом, кратным 16. Это адрес 10410h. С него ассемблер и начнет размещение предложений сегмента B, заняв под них 6 байт. Сегмент C будет размещен со следующего адреса, кратного 16, т. е. с 10420h. Таким образом, ассемблер сам следит за тем, чтобы

сегменты размещались в памяти с адресов, кратных 16, избавляя нас от этой заботы.

Значение имени сегмента

Ассемблер приписывает имени каждого программного сегмента определенное значение. Им является номер соответствующего сегмента памяти, т. е. первые 16 бит (первые 4 шестнадцатеричные цифры) начального адреса данного сегмента. В нашем примере имени A будет приписано значение 1000h, имени B – значение 1041h, а имени C – значение 1042h. Встречая в тексте программы имя сегмента, ассемблер будет заменять его на данную величину. Например, команда

```
MOV AX,B
```

будет восприниматься ассемблером как команда

```
MOV AX,1041h    ;AX:=номер сегмента B
```

Отметим особо, что в ЯА имена сегментов отнесены к константным выражениям, а не к адресным. Поэтому наша команда записывает в регистр AX число 1041h, а не содержимое ячейки по такому адресу. (По этой же причине команда LEA AX,B будет восприниматься как ошибочная, т. к. в LEA нельзя указывать константу.)

Операторы OFFSET и SEG

При трансляции предложений сегмента ассемблер ставит в соответствие описанным в нем именам переменных и меткам адреса – смещения соответствующих ячеек, отсчитанные от начала сегмента. В нашем примере имя A1 получит смещение 0, имя A2 – смещение 400h, имя B1 – смещение 0, имя B2 – смещение 2, имя L – смещение 0. Именно на эти адреса (смещения) ассемблер и заменяет имена, когда встречает их в тексте программы.

При этом, напомним, имена переменных и метки рассматриваются в ЯА как адресные выражения. Но в языке есть возможность рассматривать их и как константные выражения, для чего используется оператор OFFSET (смещение):

```
OFFSET <имя>
```

Значением оператора является смещение указанного имени, отсчитанное от начала того сегмента, в котором оно описано. Поэтому в нашем примере:

```
OFFSET A1 = 0,  OFFSET A2 = 400h,  OFFSET B1 = 0
```

При этом значение данного оператора рассматривается как константа, а не адрес. Поэтому имеем:

```
MOV AX,A2          ;AX:=содержимое ячейки A2 (AX:=8)
MOV AX,OFFSET A2   ;AX:=смещение A2 (AX:=400h) = LEA AX,A2
```

Итак, A2 – это адрес, а OFFSET A2 – это константа, хотя по величине они совпадают.

Рассмотрим пример, где полезен этот оператор. Пусть в регистре BX находится адрес некоторого элемента массива A1, т. е. BX=A1+i, и надо в BX записать индекс этого элемента, т. е. i. Сделать это командой SUB BX,A1 нельзя, т. к. она вычтет из BX содержимое начального элемента массива A1, а не адрес этого элемента (к тому же здесь будет вычитание байта из слова). Но вот команда SUB BX,OFFSET A1 правильно решит нашу проблему, т. к. из BX вычтет именно адрес A1: BX:=BX-адрес(A1)=i.

В ЯА есть также оператор, который позволяет узнать, в каком сегменте описано имя переменной или метка. Это оператор SEG (сегмент):

```
SEG <имя>
```

Его значением является имя того программного сегмента, в котором описано имя, указанное как операнд, а точнее – значение имени этого сегмента, т. е. номер соответствующего сегмента памяти. Поэтому в нашем примере:

```
SEG A1 = SEG A2 = A = 1000h
SEG B1 = SEG B2 = B = 1041h
SEG L = C = 1042h
```

При этом значение оператора SEG считается константой:

```
MOV BX,SEG B1   ;BX:=B=1041h (но не BX:=0)
```

Адресные переменные

Теперь уточним, как ассемблер обрабатывает описание переменных, в качестве начальных значений которых указаны адресные выражения. Такие переменные описываются с помощью директив DW и DD. В нашем примере в качестве начального значения переменных B1 и B2 указано имя переменной A2, т. е. адрес этой переменной. Но какой адрес – абсолютный или относительный (смещение) – здесь имеется в виду? Правила здесь такие: если имя переменной или метки (либо адресное выражение, содержащее такое имя или метку) указано как операнд директивы DW, то ассемблер заменяет имя на его смещение, но если имя указано в директиве DD, тогда ассемблер заменяет его на адресную пару, состоящую из номера сегмента, где описано это имя, и из смещения имени внутри данного сегмента (эта адресная пара определяет абсолютный адрес имени):

```
B1 DW A2   эквивалентно B1 DW offset A2
B2 DD A2   эквивалентно B2 DD seg A2 : offset A2
```

Таким образом, хотя в директиве DD мы указываем только имя и не указываем никакого сегмента, ассемблер сам определяет этот сегмент и присоединяет его к смещению. При этом, в силу "перевернутого" представления двойных слов в памяти ПК, часть seg этой пары оказывается во втором слове данного двойного слова, а часть offset – в первом слове (B2: offset, B2+2: seg).

Отметим также, что в директиве DD как операнд можно указать адресную пару следующего вида:

```
<имя сегмента>:<адресное выражение>
```

Пример:

```
DD A:B2
```

В этом случае ассемблер заменяет имя сегмента на его номер, а вот имя переменной заменяет на его смещение, но отсчитанное не от начала того сегмента, где оно описано, а от начала указанного сегмента. Например, в нашей директиве имя A будет заменено на 1000h, а имя B2 – на 412h, т. к. абсолютный адрес переменной B равен 10412h, а абсолютный адрес начала сегмента A равен 10000h: 10412h-10000h=412h. И хотя подобные операнды допустимы, ими надо пользоваться осторожно, т. к. расстояние от начала сегмента A до переменной B2 из другого сегмента может быть слишком большим и превзойти максимально допустимую величину (64 Кб) для смещений. Кроме того, как мы увидим в гл. 12, при трансляции программы возможна перестановка ее сегментов, и может случиться так, что сегмент A окажется расположенным в памяти после сегмента B, где описана переменная B2, и тогда смещение получится отрицательным, что, конечно, недопустимо.

7.3. Директива ASSUME

Итак, если мы объединили какие-то предложения в программный сегмент, то ассемблер разместит их в одном сегменте памяти. Это значит, что все его ячейки можно сегментировать по одному и тому же сегментному регистру. По какому именно? Это определяет автор программы. Скажем, в нашем примере мы можем выбрать для сегмента A регистр ES, а для сегмента B регистр DS. О своем выборе мы должны сообщить ассемблеру, чтобы он мог транслировать команды программы в соответствии с нашим решением. Дело в том, что если в команде перед операндом из памяти явно указан префикс (как, например, в команде MOV AX,ES:A2), тогда с этим префиксом ассемблер и сформирует соответствующую машинную команду. Однако выписывать всякий раз префикс накладно, поэтому хотелось бы, чтобы мы сами не указывали префикс (например, записывали бы предыдущую команду как MOV AX,A2) и чтобы ассемблер за нас "дописывал" нужный префикс. Так вот, чтобы ассемблер мог это сделать, он и должен знать, какой сегментный регистр мы выбрали для какого сегмента.

Информация о соответствии между сегментными регистрами и программными сегментами сообщается ассемблеру с помощью директивы ASSUME (предполагать):

```
          ASSUME <пара> {, <пара>}
или       ASSUME NOTHING
где <пара> – это <сегментный регистр>:<имя сегмента>
          либо <сегментный регистр>:NOTHING
```

Так, указанной в нашем примере директивой

```
ASSUME ES:A, DS:B, CS:C
```

мы сообщаем ассемблеру, что для сегментирования адресов из сегмента A мы выбрали регистр ES, для адресов из сегмента B – регистр DS, а для адресов из сегмента C – регистр CS, и требуем от ассемблера, чтобы в командах, где префикс не указан, он все имена из сегмента A транслировал с префиксом ES:, имена из сегмента B – с префиксом DS:, а имена из сегмента C – с префиксом CS:. В связи с этим команду MOV AX,A2 из нашего примера ассемблер будет воспринимать как сокращение команды MOV AX,ES:A2, поскольку имя A2 описано в сегменте A, а на него, согласно директиве ASSUME, указывает регистр ES, и потому будет транслировать ее с префиксом ES:. Команда же MOV AX,B2 будет рассматриваться как сокращение команды MOV AX,DS:B2, ибо для сегмента B, где описано имя B2, мы выбрали регистр DS, и потому команда может транслироваться как с префиксом DS:, так и без него, поскольку в этой команде префикс DS: подразумевается по умолчанию (ассемблер выбирает вариант без префикса, т. к. он экономнее по памяти).

Итак, если с помощью директивы ASSUME мы сообщим ассемблеру о соответствии между сегментными регистрами и сегментами, то получим право не указывать в командах (по крайней мере, в большинстве из них) префиксы – опущенные префиксы будет самостоятельно восстанавливать ассемблер. Как он это делает и в каких случаях он не может восстановить префиксы, будет подробно рассмотрено чуть позже, а пока отметим некоторые особенности самой директивы ASSUME.

Особенности директивы ASSUME

Прежде всего отметим, что директива ASSUME не загружает в сегментные регистры начальные адреса сегментов (такая загрузка делается отдельно – см. следующий раздел). Этой директивой автор программы лишь обещает, что в программе будет сделана такая загрузка.

Директиву ASSUME можно размещать в любом месте программы, но обычно ее указывают в начале сегмента команд. Дело в том, что информация из нее нужна только при трансляции команд, поэтому до сегмента команд эта информация еще не нужна, а при трансляции первой командой, где указано какое-либо имя, она уже нужна. Поэтому начало сегмента команд – как раз то место, где и имеет смысл размещать эту директиву.

При этом в директиве обязательно должно быть указано соответствие между регистром CS и данным сегментом команд, иначе при появлении первой же метки ассемблер зафиксирует ошибку.

Если в директиве ASSUME указано несколько пар с одним и тем же сегментным регистром, то последняя из них "отменяет" предыдущие, т. к. каж-

дому сегментному регистру можно поставить в соответствие только один сегмент. Например, по директиве

```
ASSUME ES:A, ES:B
```

ассемблер будет считать, что регистр ES указывает только на сегмент B.

В то же время на один и тот же сегмент могут указывать разные сегментные регистры. Например, согласно директиве

```
ASSUME ES:A, DS:A
```

считается, что на сегмент A указывает как регистр ES, так и регистр DS. В данном случае ассемблер получает свободу в выборе префикса, с которым он должен транслировать имена из сегмента A. Как мы увидим чуть позже, в такой ситуации ассемблер отдает предпочтение, если можно, тому префиксу, который в транслируемой команде подразумевается по умолчанию, поскольку его можно опустить и на этом сэкономить память.

Если в директиве ASSUME в качестве второго элемента пары задано служебное слово NOTHING (ничего), например ASSUME ES:NOTHING, то это означает, что с данного момента сегментный регистр не указывает ни на какой сегмент, что ассемблер не должен использовать этот регистр при трансляции команд и что автор программы берет на себя обязанность, когда надо, явно указывать этот регистр. Если же нужно отметить ранее установленные соответствия для всех сегментных регистров, тогда используется директива ASSUME NOTHING.

В программе директива ASSUME может быть указана любое число раз. При этом несколько подряд идущих директив можно объединить в одну. Например, последовательность директив

```
ASSUME ES:A
ASSUME DS:B, CS:C
```

эквивалентна одной директиве

```
ASSUME ES:A, DS:B, CS:C
```

Однако, если между соседними директивами имеются команды, то объединение директив не всегда возможно. Например, во фрагменте (имя A2 описано в сегменте A)

```
ASSUME ES:A
MOV CX,A2
ASSUME ES:B, SS:A
INC A2
```

имя A2 в команде MOV будет оттранслировано с префиксом ES:, а в команде INC – с префиксом SS:, тогда как во фрагменте

```
ASSUME ES:A, ES:B, SS:A
MOV CX,A2
INC A2
```

в обеих командах имя A2 будет оттранслировано в префиксом SS:.

Выбор сегментных регистров при трансляции команд

Теперь сформулируем правила, по которым ассемблер при трансляции команды выбирает префикс для операнда из памяти в том случае, когда префикс не указан явно.

Если в записи операнда нет имени, по которому ассемблер мог бы определить, в каком сегменте памяти расположен операнд, тогда ассемблер выбирает тот сегментный регистр, который в этой команде подразумевается по умолчанию. Например, в команде MOV AX,[BX] будет выбран префикс DS:, а в команде MOV AX,[BP] – префикс SS:.

Если же в операнде указано имя переменной или метка, тогда ассемблер смотрит, было ли это имя уже описано в программе. Если нет, если это ссылка вперед, тогда ассемблер предполагает, что это имя должно сегментироваться по регистру, который в этой команде подразумевается по умолчанию. Например, если X – ссылка вперед, то в команде INC X или INC X[SI] будет подразумеваться префикс DS;, а в команде INC X[BP+1] – префикс SS:. Но если затем обнаружится, что это предположение неверно, тогда будет зафиксирована ошибка, т. к. "задним числом" ассемблер не сможет вставить пропущенный префикс – для него нет уже места.

Если в операнде транслируемой команды указано имя и оно уже описано, тогда ассемблер определяет по директиве ASSUME, какие сегментные регистры поставлены в соответствие сегменту, в котором описано это имя. Если таких регистров нет, ассемблер фиксирует ошибку. В противном случае из этих регистров ассемблер выбирает регистр, подразумеваемый в команде по умолчанию, а если такого нет, выбирает любой другой.

Установив, с каким префиксом надо транслировать операнд, ассемблер затем смотрит, не совпадает ли он с префиксом, подразумеваемым в команде по умолчанию. Если не совпадает, ассемблер формирует машинную команду с этим префиксом, а если совпадает – без префикса, т. к. он попросту лишний.

Таковы правила, по которым ассемблер определяет, с каким префиксом он должен транслировать операнд из памяти, если префикс не указан явно. Как видно, они достаточно запутанны. Однако в реальных программах в большинстве команд в качестве операнда указывается имя, которое уже описано в программе и сегменту которого по директиве ASSUME поставлен в соответствие определенный сегментный регистр, поэтому в таких командах ассемблер самостоятельно и правильно подберет нужный префикс. И только в небольшом числе случаев приходится явно указывать префикс. Эти случаи таковы (они вытекают из рассмотренных правил).

Первый случай связан с косвенными ссылками. Например, при трансляции команды ADD DX,[BX] ассемблер выберет префикс DS:, подразумеваемый в ней по умолчанию, т. е. будет считать, что в регистре BX находится ссылка в сегмент данных. Если нас это не устраивает, если мы знаем, что в BX находится ссылка в другой сегмент, например в сегмент команд, тогда мы обязаны явно указать нужный префикс: ADD DX,CS:[BX].

Второй случай связан со ссылками вперед, т. е. когда в команде указывается имя, которое будет описано позже. Ассемблер в такой ситуации использует сегментный регистр, подразумеваемый в команде по умолчанию. Если нас это не устраивает, если мы заранее знаем, что это имя описывается в сегменте, на который указывает иной сегментный регистр, тогда мы обязаны явно указать этот иной регистр. Например, в следующем фрагменте

```
CODE SEGMENT
     ASSUME CS:CODE, ES:DATA1
     MOV AX,ES:X
      ...
CODE ENDS
DATA1 SEGMENT
    X DW 23
      ...
DATA1 ENDS
```

в команде MOV обязательно надо записать префикс ES:, т. к. без него ассемблер оттранслирует команду без префикса, а затем, встретив описание имени X в сегменте DATA1, на который установлен регистр ES, зафиксирует ошибку. Однако в данной ситуации лучше действовать по-иному: чтобы не попадаться на подобные ошибки со ссылками вперед и чтобы пореже выписывать префиксы перед именами, следует сегменты данных размещать перед сегментом команд, тогда попросту не будет ссылок вперед. Другими словами, рекомендуется размещать сегмент команд в самом конце программы.

Третий случай – это когда некоторому сегменту мы не поставили в соответствие (в директиве ASSUME) никакой сегментный регистр, и потому ассемблер, не имея нужной информации, не может самостоятельно определить, с каким префиксом надо транслировать имена из этого сегмента (считается, что в этом случае автор программы берет на себя ответственность указывать такой префикс). Подобной ситуации лучше избегать, если мы не хотим перед каждым именем из этого сегмента явно указывать префикс.

И, наконец, еще один случай, когда приходится выписывать префикс, связан с заданием явных адресов. Если, к примеру, в регистр AX надо записать содержимое ячейки с адресом 5, то сделать это командой MOV AX,5 нельзя. Причин здесь две. Одна чисто формальная: в ЯА явно указанное число всегда трактуется как константа (непосредственный операнд), а не как адрес. Поэтому наша команда определяет операцию AX:=5. Другая причина более важная: даже если считать 5 адресом, остается непонятным, из какого сегмента памяти должна браться ячейка с этим адресом, по какому регистру надо сегментировать этот адрес. В ЯА проблема с явными адресами решается следующим образом: перед такими адресами обязательно надо указывать префиксы. Например, если мы имеем в виду 5-ю ячейку из сегмента, на начало которого указывает регистр DS, тогда мы должны записать команду в таком виде: MOV AX,DS:5. Этим префиксом мы сразу "убиваем двух зайцев":

во-первых, префикс указывает, что 5 надо воспринимать не как число, а как адрес, и во-вторых, он решает проблему с сегментированием адреса.

Таковы случаи, когда в программах на ЯА приходится явно указывать префиксы. Но, к счастью, они встречаются редко, и в подавляющем большинстве команд префиксы можно не указывать.

В заключение отметим еще одну особенность трансляции операндов из памяти, но уже возникающую в том случае, когда перед операндом явно указан префикс. Рассмотрим следующий пример:

```
DATA SEGMENT
   X DW ?
     ...
DATA ENDS
CODE SEGMENT
     ASSUME CS:CODE, ES:DATA
   Y DW ?
  L: MOV ES:Y,0
     ...
```

С одной стороны, в команде MOV должен использоваться префикс ES:, который явно указан, но с другой стороны, имя Y описано в сегменте CODE, на начало которого установлен регистр CS, и потому должен использоваться префикс CS:. С каким же префиксом надо транслировать эту команду? В такой ситуации ассемблер поступает следующим образом: он транслирует команду с указанным префиксом (у нас – с ES:), а вместо имени Y подставляет его смещение, но отсчитанное не от начала его "родного" сегмента CODE, а от начала того сегмента, на который указывает регистр ES – от начала DATA. И если в регистре ES, как пообещал автор программы, действительно находится начальный адрес сегмента DATA, тогда сформированная таким образом машинная команда будет работать правильно, будет работать именно с переменной Y, а не с X, как было бы, если бы имя Y заменялось на его смещение (0), отсчитанное от начала сегмента CODE. (Отметим, однако, что если нет информации о том, на какой сегмент установлен регистр ES, тогда ассемблер заменит имя Y на смещение, отсчитанное от начала его "родного" сегмента CODE, сохранив при этом префикс ES:.)

Подобного рода записи операндов надо избегать. Дело в том, что расстояние от начала сегмента DATA до переменной Y может быть слишком большим, поэтому смещение имени Y, отсчитываемое от начала DATA, может превзойти максимально допустимую величину 64 Кб. Кроме того, как будет рассказано в гл. 12, при трансляции программы возможна перестановка ее сегментов, и может оказаться так, что сегмент DATA будет расположен в памяти после сегмента CODE, и тогда смещение для Y окажется отрицательным, что, конечно, недопустимо. Поэтому надо выбрать одно из двух – либо полностью возложить на ассемблер ответственность за выбор префиксов (что

обычно и делают), либо самому явно указывать префиксы. Смешивать же эти две линии поведения не следует.

7.4. Начальная загрузка сегментных регистров

Итак, директивой ASSUME мы сообщаем, что все имена из таких-то программных сегментов должны сегментироваться по таким-то сегментным регистрам, и требуем, чтобы ассемблер формировал машинные команды с соответствующими префиксами. Однако это еще не значит, что программа будет работать правильно. Скажем, у нас по команде MOV AX,A2 будет сформирована команда MOV AX,ES:A2, но если регистр ES не будет указывать на начало сегмента A, то такая команда проработает неправильно. Нужно еще, чтобы в сегментных регистрах находились правильные значения – номера соответствующих сегментов памяти. Но в начале выполнения программы в этих регистрах ничего нет. Поэтому выполнение программы надо начинать с команд, которые загружают в сегментные регистры номера соответствующих сегментов памяти (напомним, что директива ASSUME такую загрузку не осуществляет). Рассмотрим, как делается такая загрузка.

Пусть регистр DS надо установить на начало сегмента B. Напомним, что имени сегмента в ЯА приписывается значение – первые 16 бит начального адреса этого сегмента, а это как раз то, что должно быть записано в сегментный регистр, поэтому нам надо выполнить присваивание DS:=B. Однако сделать это командой MOV DS,B нельзя, поскольку имя сегмента – это константное выражение, т. е. непосредственный операнд, а по команде MOV запрещена пересылка непосредственного операнда в сегментный регистр. Поэтому такую пересылку надо делать через другой, несегментный регистр, например, через AX:

```
MOV AX,B
MOV DS,AX  ;DS:=B
```

Аналогично загружается и регистр ES.

Что касается регистра CS, то его загружать не надо. Дело в том, что к началу выполнения программы этот регистр уже будет указывать на начало сегмента команд. Такую загрузку сделает операционная система, прежде чем передаст управление на программу, поэтому загружать регистр CS в самой программе не надо.

Теперь о регистре SS, который должен указывать на начало стека. Загрузить этот регистр можно двояко. Во-первых, его можно загрузить в самой программе – так же, как мы загружаем регистры DS и ES. Во-вторых, такую загрузку можно поручить операционной системе. Для этого в директиве SEGMENT, открывающей описание сегмента стека, надо указать специальный параметр STACK, например:

```
S SEGMENT STACK
```

В таком случае загрузка S в регистр SS будет выполнена автоматически до начала выполнения программы. Ясно, что из этих двух вариантов загрузки регистра SS лучше второй, поэтому в дальнейшем при описании сегмента стека мы всегда будем указывать параметр STACK.

7.5. Структура программы. Директива INCLUDE

Теперь мы имеем достаточно информации и, наконец-то, можем рассказать о том, как выглядит полная программа на ЯА. Отметим, что какой-либо фиксированной структуры программы на ЯА нет, но для небольших программ, в которой используется три сегмента – команд, данных и стека, типичной является такая структура:

```
 STACK SEGMENT STACK    ;сегмент стека
       DB 128 DUP(?)
 STACK ENDS

  DATA SEGMENT          ;сегмент данных
       <описания переменных и т. п.>
  DATA ENDS

  CODE SEGMENT          ;сегмент команд
       ASSUME CS:CODE,DS:DATA,SS:STACK
START: MOV AX,DATA
       MOV DS,AX        ;загрузка DS
       <остальные команды программы>
  CODE ENDS
       END START        ;конец программы, точка входа
```

Поясним эту структуру программы.

Взаимное расположение сегментов программы может быть любым, но как уже отмечалось, чтобы сократить в командах число ссылок вперед и избежать проблем с префиксами для них, рекомендуется сегмент команд размещать в конце текста программы.

Сегмент стека в нашей программе описан с параметром STACK (совпадение этого параметра с именем сегмента допустимо), поэтому в самой программе нам уже не надо загружать сегментный регистр SS. Не надо, как уже отмечалось, загружать и регистр CS. Поэтому в начале программы, как видно, загружается лишь регистр DS.

Относительно сегмента стека нужно сделать следующее замечание. Даже если сама программа не использует стек, описывать в программе сегмент стека все равно надо. Дело в том, что стек программы использует операционная система при обработке прерываний, возникающих (например, при нажатии клавиш на клавиатуре) в процессе счета программы, о чем будет рассказано в гл. 13. А пока лишь отметим, что в такой ситуации рекомендуемый размер стека (с запасом) – 128 байт.

В конце программы обязательно надо указать директиву END. Она является признаком конца текста программы, все строки за ней считаются уже не

относящимися к программе. Кроме того, в этой директиве указывается так называемая точка входа – метка той команды, с которой должно начаться выполнение программы (в общем случае это необязательно первая команда программы). У нас это метка START.

Теперь сделаем несколько замечаний общего характера.

Все предложения, по которым ассемблер что-то заносит в формируемую им машинную программу (а это, в первую очередь, команды и директивы определения данных), обязательно должны входить в состав какого-то программного сегмента, размещать их вне программных сегментов нельзя. Вне программных сегментов можно указывать только директивы чисто информационного характера, например, директивы EQU, директивы описания типов структур и записей.

Может возникнуть вопрос: разрешается ли в сегменте данных размещать команды, а в сегменте команд размещать описания переменных? Да, можно, но лучше это не делать, т. к. будут проблемы с сегментированием. Поэтому рекомендуется в сегменте данных описывать только переменные, а в сегменте команд размещать только команды.

Директива INCLUDE

В данной книге мы договорились пользоваться нестандартными операциями ввода-вывода (их реализация будет рассмотрена в гл. 13). Будем считать, что описание этих операций находится в файле с названием IO.ASM. Так вот, чтобы мы могли в программе пользоваться этими операциями, данное описание надо вставить в текст нашей программы. Для этого следует в первой строчке нашей программы указать директиву

```
INCLUDE IO.ASM
```

В общем случае обращение к директиве INCLUDE (включить) имеет такой вид:

```
INCLUDE <имя файла>
```

Встречая эту директиву, ассемблер весь текст, хранящийся в указанном файле, подставит в программу вместо этой директивы.

Директиву INCLUDE можно указывать любое число раз и в любых местах программы. В ней можно указать любой файл, причем название файла записывается по правилам операционной системы, например:

```
INCLUDE A:MACROS.TXT
```

Директива полезна, когда в разных программах используется один и тот же фрагмент текста; чтобы не выписывать его в каждой программе заново, его записывают в какой-то файл, а затем подключают к программам с помощью данной директивы. В нашем случае вместо директивы в текст программы подставится описание операций ввода-вывода, и тем самым мы получаем право пользоваться ими в нашей программе.

В заключение данного раздела приведем пример простой, но полной программы на ЯА. Эта программа вводит 50 символов и выводит их в обратном порядке.

Для решения этой задачи заведем массив из 50 байт и будем записывать в него в обратном порядке (от конца к началу) вводимые символы, а в конце выведем его содержимое просто как строку.

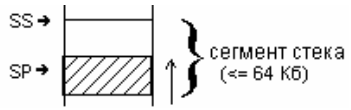
```
     INCLUDE IO.ASM    ;подключение операций ввода-вывода
   S SEGMENT STACK     ;сегмент стека
     DB 128 DUP(?)
   S ENDS
   D SEGMENT           ;сегмент данных
   N EQU 50            ;число символов для ввода
   X DB N DUP(?), '$'  ;строка для записи символов + конец
                       ;строки
   D ENDS
   C SEGMENT           ;сегмент команд
     ASSUME SS:S, DS:D, CS:C
BEG: MOV AX,D
     MOV DS,AX
     OUTCH '>'         ;приглашение к вводу
     MOV CX,N          ;счетчик цикла
     MOV SI,N-1        ;индекс (=49, 48, ...)
 IN: INCH X[SI]        ;ввод символа и запись его в конец X
     DEC SI
     LOOP IN
     LEA DX,X          ;вывод строки X
     OUTSTR
     FINISH
   C ENDS
     END BEG           ;точка входа – команда BEG
```

8. СТЕК

В данной главе рассматриваются стековые команды и приемы работы со стеком.

8.1. Стек и сегмент стека

Стек – это хранилище, работа с которым ведется по следующему принципу: элемент, записанный в стек последним, считывается из него первым. В ПК для такого хранилища можно отвести любую область памяти, но к ней предъявляется два требования: ее размер не должен превышать 64 Кб и ее начальный адрес должен быть кратным 16. Другими словами, эта область должна быть сегментом памяти. Его называют сегментом стека. На его начало, как мы уже знаем, должен указывать сегментный регистр SS:



В ПК принято заполнять стек снизу вверх: первый элемент записывается в самый конец области стека (в ячейку области с наибольшим адресом), следующий элемент записывается "над" ним и т. д. При чтении же из стека первым всегда удаляется самый верхний элемент. Поэтому получается, что низ стека фиксирован (это последняя ячейка области стека), а вот вершина стека все время сдвигается. Для того чтобы знать текущее положение этой вершины, используется еще один регистр – SP (stack pointer, указатель стека). В нем хранится адрес той ячейки, в которой находится элемент, записанный в стек последним. Более точно, в SP находится смещение этой ячейки, т. е. ее адрес, отсчитанный от начала сегмента стека. Поэтому абсолютный адрес вершины стека задается парой регистров SS:SP.

Элементы стека могут иметь любой размер; это могут быть байты, слова, двойные слова и т. д. Однако имеющиеся в ПК команды записи в стек и считывания из него работают только со словами. Поэтому обычно подстраиваются под эти команды и считают, что элементы стека имеют размер слова. Обработка же байтов и двойных слов подгоняется под обработку слов.

Следует различать термины "сегмент стека" и "стек (содержимое стека)": если первый термин означает область памяти, которую потенциально могут занять данные стека, то второй термин обозначает совокупность тех данных, что в текущий момент хранятся в стеке, т. е. совокупность байтов от адреса из SP до конца сегмента стека. Причем все данные, расположенные "выше" адреса из SP, считаются не относящимися к стеку.

Чтобы зарезервировать место в памяти под стек, в программе на ЯА следует описать соответствующий программный сегмент. Если мы решили выделить под него k байт, тогда этот сегмент должен описываться так:

```
S SEGMENT STACK
  DB k DUP(?)
S ENDS
```

Имена ячейкам стека обычно не дают, т. к. доступ к ним все равно будет осуществляться не по именам, а косвенно, через регистр SP. Чаще всего не задают и начальные значения для ячеек стека. Поэтому при описании сегмента стека указывают лишь то, сколько байтов отводится под стек, т. е. используют безымянную директиву DB со знаком ?. Отметим, что вместо директивы DB можно использовать директиву DW или DD, это как кому нравится.

Прежде чем начать работу со стеком, надо загрузить в регистры SS и SP соответствующие значения: регистр SS должен указывать на начало сегмента стека, а регистр SP – на вершину стека. Сделать это можно двояко. Во-первых, такую загрузку мы можем сделать сами, поместив в программу соответствующие команды. Во-вторых, если в директиве SEGMENT, начинающей описание сегмента стека, указать параметр STACK, тогда к началу выполнения программы оба этих регистра будут загружены автоматически: регистр SS будет установлен на начало сегмента стека, а в регистр SP будет записан размер стека в байтах (число k). Почему именно это значение? В начале работы программы стек должен быть, как правило, пустым, а значение k, как несложно сообразить, как раз и соответствует пустому стеку; в этом случае SP указывает на первую ячейку за областью стека.

И, наконец, напомним, что сегмент стека надо описывать в программе даже тогда, когда программа сама его не использует. В таком случае рекомендуемый размер (с запасом) сегмента стека – 128 байт. Если же программа сама использует стек, то под него надо отводить столько места, сколько надо программе, плюс эти 128 байт.

8.2. Стековые команды

Для работы со стеком в ПК имеется несколько команд, которые принято называть стековыми. Сразу отметим, что все они будут работать правильно, только если регистр SS указывает на начало сегмента стека, а регистр SP – на вершину стека. Если в этих регистрах записано что-то иное, то действие стековых команд непредсказуемо.

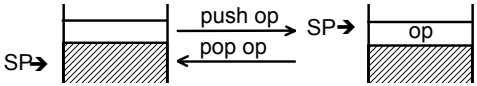
8.2.1. Запись и чтение слов

Основными стековыми командами ПК являются команды записи слова в стек и считывания слова из стека.

Запись слова в стек: PUSH op

Допустимые типы операнда: r16, sr, m16.

Команда PUSH ("вталкивать") записывает в стек свой операнд. Условно это можно изобразить так:



Более точно команда PUSH действует так: сначала значение регистра SP уменьшается на 2 (вычитание происходит по модулю 2^{16}), т. е. SP сдвигается вверх и теперь указывает на свободную ячейку области стека, а затем в нее записывается операнд:

`SP:=(SP-2) mod 2^16, op -> [SS:SP].`

Флаги команда не меняет.

Замечание. По команде PUSH SP процессоры 8086 и 80286 в стек записывают значение SP после изменения этого регистра, а процессоры i386 и i486 – значение SP до изменения.

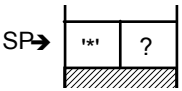
В команде PUSH в качестве операнда можно указывать любой сегментный регистр (например, PUSH CS), но нельзя указывать непосредственный операнд (например, PUSH 5 – ошибка). Если мы хотим записать в стек явное число, то это надо делать через регистр, например:

```
MOV AX,5
PUSH AX    ;5 -> стек
```

Отметим также, что по команде PUSH в стек можно записать только слово, но не байт или двойное слово. Двойное слово приходится записывать двумя командами, а для записи байта (скажем, символа '*') этот байт расширяют каким угодно способом до слова и в стек уже записывают слово:

```
MOV AL,'*' ; AL:='*' (AH:=?)
PUSH AX
```

При этом в силу "перевернутого" представления слов в памяти ПК код символа '*' займет в стеке байт, на который указывает регистр SP:



Конечно, при таком способе записи байта в стек, мы теряем байт, но обычно на это идут, если хотят воспользоваться командой PUSH. Для чтения же байта из стека надо считать все слово, а затем из него выделить нужный байт.

Чтение слова из стека: POP op

Допустимые типы операнда: r16, sr (кроме CS), m16.

Команда POP ("выталкивать") считывает слово из вершины стека и присваивает его указанному операнду (см. рис. выше). Более точно: слово из ячейки, на которую указывает пара SS:SP, пересылается в операнд, а затем

SP увеличивается на 2 (сложение происходит по модулю 2^{16}), т. е. сдвигается вниз:

```
[SS:SP] -> op, SP:=(SP+2) mod 2^16.
```

Слово, считанное из стека по команде POP, может быть помещено в любой регистр, кроме сегментного регистра CS (изменение этого регистра означает переход). Кроме того, по этой команде можно считать только слово, но не байт или двойное слово.

8.2.2. Запись и чтение регистра флагов

Следующая пара стековых команд используется для записи в стек и чтения из стека регистра флагов.

Запись регистра флагов в стек: `PUSHF`
Чтение регистра флагов из стека: `POPF`

По команде PUSHF в стек записывается содержимое регистра флагов Flags, а по команде POPF из стека считывается слово и оно заносится в Flags. При этом команда PUHSF не меняет флаги, а команда POPF, естественно, меняет все флаги.

Эти команды обычно используются для сохранения текущих состояний флагов и последующего их восстановления. Эти команды можно также использовать для определения или изменения состояния любого флага – в том случае, если нет подходящих команд для реализации этих действий. Например, записать в регистр AX значение флага трассировки TF (он занимает 8-й бит регистра Flags при нумерации битов справа налево от 0 – см. разд. 1.2.4), не меняя при этом никакие флаги, можно так:

```
PUSHF      ;запомнить Flags, чтобы затем восстановить его
PUSHF      ;запомнить Flags для пересылки в AX
POP AX     ;AX:=Flags
MOV CL,8
SHR AX,CL  ;сдвиг бита с TF к правому краю AX
AND AX,1b  ;AX:=TF
POPF       ;восстановить исходное значение Flags
```

8.2.3. Стековые команды процессора 80186

В процессоре 80186 появился вариант команды PUSH, в котором допускается непосредственный операнд. Поэтому запись в стек явного числа (например, 5) в ПК с таким или более старшим процессором реализуется проще, чем в ПК с процессором 8086:

```
.186
...
PUSH 5
```

(Напомним, что по умолчанию ассемблер разрешает указывать в программах на ЯА только команды процессора 8086, и для того, чтобы можно было использовать команды процессора 80186, надо перед первой из таких команд задать директиву .186)

В процессоре 80186 появились еще две стековые команды – запись в стек и чтение из стека значений всех регистров общего назначения. Флаги эти команды не меняют.

Запись регистров в стек: `PUSHA`
Чтение регистров из стека: `POPA`

Команда PUSHA записывает в стек значения всех регистров общего назначения в следующей последовательности: AX, CX, DX, BX, SP, BP, SI и DI (для SP берется значение до этой команды). Команда POPA считывает из стека 8 слов и присваивает их регистрам в последовательности, обратной указанной выше (после команды регистр SP указывает на состояние стека после этого считывания).

Команды PUSHA и POPA предназначены для работы с процедурами (см. гл. 9) – для сохранения значений регистров при входе в процедуру и восстановления прежних значений при выходе из процедуры.

8.3. Некоторые приемы работы со стеком

Прежде всего отметим общее правило работы со стеком: если мы что-то записали в стек, то обычно именно мы и обязаны все это считать из стека. Иначе при работе со стеком "не найдешь концов".

Сохранение значений регистров

Очень часто стек используется для временного хранения значений регистров. Если нам надо сохранить текущее значение какого-нибудь регистра, скажем CX, и в то же время этот регистр нам нужен для иных целей, тогда можно поступить так: надо спасти в стеке значение регистра, затем использовать регистр как нужно, а в конце восстановить прежнее значение регистра, считав его из стека:

```
PUSH CX
....        ;использование CX
POP CX
```

Пересылка данных через стек

Стек нередко используется для пересылки какой-то величины из одной ячейки памяти в другую, когда не хотят портить содержимое регистров. Например, выполнить присваивание X:=Y, где X и Y – переменные размером в слово, можно так:

```
PUSH Y
POP X       ;X:=Y
```

Проверка на выход за пределы стека

Следует понимать, что команды PUSH и POP не осуществляют проверку на выход за пределы стека. Например, если стек пуст и мы применяем команду чтения из стека, то ошибка не будет зафиксирована (будет считано слово, следующее за сегментом стека). Аналогично, не будет зафиксирована

ошибка, если мы записываем в стек, когда он уже полон. Такие проверки, если надо, обязаны делать мы сами. Делаются же эти проверки очень просто:

```
SP=0? – стек полон?
SP=k? – стек пуст? (k – размер сегмента стека в байтах)
```

В самом деле, стек полон, если вершина стека достигла начала области, выделенной для стека, а это значит, что смещение вершины стека равно 0. При пустом же стеке, как уже отмечалось, в регистре SP находится число, равное размеру области стека в байтах.

Очистка и восстановление стека

Иногда приходится очищать стек, т. е. удалять из него несколько последних элементов, никуда их не переписывая. Конечно, здесь можно выполнить нужное число раз команду POP, но это долго. Очистку стека можно сделать и проще, если заметить, что после удаления из стека N слов значение SP должно увеличиться на число 2*N, и если вспомнить, что регистр SP, хотя и специализирован для работы со стеком, все-таки является регистром общего назначения, который можно использовать в любых командах, в том числе и в арифметических. Поэтому очистка стека от N слов осуществляется просто увеличением значения регистра SP на величину 2*N:

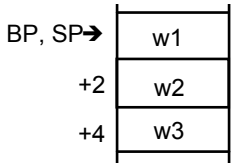
```
ADD SP,2*N ;очистка стека от N слов
```

Другой вариант очистки стека: запомнить вначале то значение указателя стека SP, до которого затем надо будет очищать стек, после чего можно записывать в стек что угодно, а в конце надо просто восстановить в SP это значение:

```
MOV AX,SP
...         ;записи в стек
MOV SP,AX
```

Доступ к элементам стека. Регистр BP

Команды PUSH и POP дают доступ только к вершине стека, но иногда необходим доступ к другим, более "низким" элементам стека. Пусть, к примеру, в стеке записано не менее трех слов и мы хотим заслать в регистр AX копию третьего сверху элемента стека (AX:=w3):



Как видно, адрес третьего слова стека равен адресу вершины стека плюс 4. Учитывая это, делаем следующее: устанавливаем регистр BP на вершину стека и используем выражение [BP+4] для доступа к третьему слову:

```
MOV BP,SP
MOV AX,[BP+4] ; = MOV AX,SS:[BP+4]
```

Почему мы использовали именно регистр BP? Во-первых, отметим, что использовать здесь регистр SP, т. е. выражение [SP+4], нельзя, т. к. SP не относится к числу регистров-модификаторов. Во-вторых, в последней из наших команд не указан никакой сегментный регистр, поэтому он выбирается по умолчанию. Напомним, что если в команде адрес модифицируется и среди модификаторов есть регистр BP, то по умолчанию выбирается регистр SS, поэтому наша команда воспринимается как MOV AX,SS:[BP+4], а это значит, что считывание происходит из ячейки сегмента стека. А вот если бы вместо BP мы использовали другой модификатор, скажем BX, тогда по умолчанию брался бы префикс DS:, что нам не подходит, и потому пришлось бы явно указывать нужный префикс: MOV AX,SS:[BX+4]. Ясно, что этот вариант хуже, чем вариант с регистром BP.

Итак, если мы хотим работать с "внутренними" элементами стека, то надо установить регистр BP на одну из ячеек стека, например на вершину стека, а затем для ссылок на элементы стека использовать конструкции вида [BP+n] с подходящим числом n. Поскольку доступ к внутренним элементам стека приходится делать довольно часто, то в ПК и выделили особо регистр BP, сделали так, чтобы по умолчанию он сегментировался по регистру SS.

8.4. Пример использования стека

В качестве примера на использование стека опишем программу, которая вводит последовательность любых символов (кроме точки) с точкой в конце и проверяет, правильно ли этот текст сбалансирован по круглым и квадратным скобкам. В качестве ответа выдается буква Д (да, текст сбалансирован) или Н (нет).

Идея решения этой задачи следующая. Вводим очередной символ и проверяем, скобка ли это. Если нет, ничего с этим символом не делаем. Если это открывающая скобка, то записываем ее в стек. Если это закрывающая скобка, тогда делаем следующие проверки. Если стек пуст (т. е. до этого в тексте не было открывающих скобок), тогда фиксируем несбалансированность. Иначе считываем из стека последнюю (именно последнюю!) из открывающих скобок и смотрим, соответствует ли она нашей закрывающей скобке: если да, то переходим к следующему символу текста, а если нет – очищаем стек от всех записанных в него открывающих скобок и фиксируем несбалансированность. По окончанию ввода текста делаем еще одну проверку: если стек в этот момент не пуст (т. е. остались открывающие скобки), то очищаем стек от этих непарных скобок и фиксируем несбалансированность, а иначе сообщаем, что текст сбалансирован по скобкам.

Теперь уточним кое-какие детали.

В этой программе нет данных, которые надо было бы хранить в памяти, поэтому в программе не надо описывать сегмент данных и загружать сегментный регистр DS. На сегмент же стека отведем обязательные 128 байт и еще 200 байт для хранения открывающих скобок (считаем, что уровень вложенности скобок в исходном тексте меньше 100).

Для проверки, пуст ли стек, и очистки стека удобно запомнить начальное значение регистра SP и затем использовать это значение для реализации указанных действий. (Перед выполнением команды FINISH не требуется очищать стек, однако для соблюдения "правил приличия" мы все же очистим стек.)

Для проверки соответствия открывающей и закрывающей скобок поступим следующим образом: при появлении закрывающей скобки заменим ее на соответствующую ей открывающую скобку и именно эту скобку будем сравнивать со скобкой, считанной из стека.

С учетом всего сказанного наша программа выглядит так:

```
        INCLUDE IO.ASM  ;подключить операции ввода-вывода
      S SEGMENT STACK;сегмент стека
        DB 328 DUP(?)
      S ENDS
      C SEGMENT         ;сегмент команд
        ASSUME CS:C, SS:S
   BEG: MOV SI,SP       ;запомнить исходное значение SP
        MOV AL,'Д'      ;предварительный ответ ("да")
        OUTCH '>'       ;приглашение к вводу
    IN: INCH BL         ;ввод очередного символа
        CMP BL,'.'
        JE ENDTXT       ;точка -> выход из цикла ввода
        CMP BL,'('      ;проверка на открыв. скобку
        JE OPEN
        CMP BL,'['
        JNE NOOPEN
  OPEN: PUSH BX         ;запомнить открыв. скобку в стеке
        JMP IN
NOOPEN: MOV BH,'('      ;проверка на закрыв. скобку
        CMP BL,')'      ;(в BH - парная ей открыв. скобка)
        JE CLOSE
        MOV BH,'['
        CMP BL,']'
        JNE IN          ;не скобка - к следующему символу
 CLOSE: CMP SP,SI       ;проверки для закрыв. скобки:
        JE NO           ;стек пуст -> NO
        POP CX          ;считать из стека открыв. скобку в CL
        CMP CL,BH       ;и сравнить ее с BH
        JE IN           ;есть парность - к следующему символу
        JMP NO          ;нет -> NO
ENDTXT: CMP SP,SI       ;в конце текста проверка
                        ;стека на "пусто"
        JE OUT
    NO: MOV SP,SI       ;нет сбалансированности - очистка
                        ;стека
        MOV AL,'Н'      ;и ответ "нет"
   OUT: OUTCH AL        ;выдача ответа
        FINISH
      C ENDS
        END BEG
```

9. ПРОЦЕДУРЫ

В данной главе рассматриваются процедуры (подпрограммы) и те проблемы, которые возникают при их реализации и использовании.

9.1. Дальние переходы

До сих пор мы рассматривали программы с одним сегментом команд, что характерно для небольших программ. Однако в общем случае в программе может быть столь много команд, что они не вместятся в один сегмент памяти (их суммарный размер может превзойти 64Кб). В таком случае (либо по какой-то иной причине) в программе описывается несколько сегментов команд. Например, в программе могут быть описаны такие сегменты команд:

```
    C1 SEGMENT
       ASSUME CS:C1, ...
START: MOV AX,0
       ...
       JMP FAR PTR L ;goto L
       ...
    C1 ENDS
    C2 SEGMENT
       ASSUME CS:C2
    L: INC BX
       ...
    C2 ENDS
```

Отметим, что в начале каждого сегмента команд должна быть указана директива ASSUME, в которой (помимо прочего) сегментному регистру CS ставится в соответствие данный сегмент (именно по этой информации ассемблер узнает, что текущий сегмент является сегментом команд). В противном случае, встретив первую же метку, ассемблер зафиксирует ошибку.

Напомним, что в ПК адрес команды, которая должна выполняться следующей, задается парой регистров CS и IP: регистр CS указывает на начало сегмента памяти, в котором находится эта команда, а в регистре IP находится смещение этой команды, отсчитанное от начала данного сегмента. Изменение любого из этих регистров есть ничто иное, как переход, поскольку меняется адрес команды, подлежащей выполнению.

Если меняется только IP, то это означает переход внутри сегмента. Такие переходы называются близкими или внутрисегментными; до сих пор только такие переходы мы и рассматривали. Если программа небольшая и все ее команды умещаются в одном сегменте памяти, то другие переходы и не нужны. Но если в программе имеется несколько сегментов команд, тогда возникает потребность в переходах из одного такого сегмента в другой (например, из сегмента C1 на метку L сегмента C2). Такие переходы называются дальними или межсегментными. При этих переходах меняется значение и регистра CS, и регистра IP: CS устанавливается на начало сегмента с меткой (CS:=C2), а в IP записывается смещение метки внутри ее сегмента (IP:=offset L).

В ПК предусмотрены команды, реализующие такие дальние переходы, причем все они являются только безусловными переходами (условные переходы – всегда близкие), прямыми или косвенными. В ЯА в этих командах указывается тот же мнемокод JMP, что и при близких переходах, но используются другие типы операндов. Флаги команды дальнего перехода не меняют.

Дальний прямой переход: JMP FAR PTR <метка>

Здесь слово FAR (дальний) указывает ассемблеру, что метка дальняя, что она находится в другом сегменте команд. По этой команде регистр CS устанавливается на начало того сегмента, в котором эта метка находится, а в регистр IP записывается смещение этой метки внутри данного сегмента:

```
CS:=seg <метка>; IP:=offset <метка>
```

Так, в примере выше указан дальний переход на метку L.

Дальний косвенный переход: JMP m32

В этой команде указывается адрес двойного слова, в котором должен находиться абсолютный адрес перехода в виде адресной пары seg:ofs, записанной в "перевернутом" виде: смещение ofs должно быть записано по адресу m32, а номер сегмента seg – по адресу m32+2. Команда делает переход по этому абсолютному адресу, т. е. записывает в регистр CS величину seg, а в регистр IP – величину ofs:

```
CS:=[m32+2]   ;IP:=[m32]
```

Пример:

```
X DD L     ;X: offset L, X+2: seg L
  ...
  JMP X    ;goto L (CS:=seg L, IP:=offset L)
  ...
```

При записи команды косвенного перехода надо соблюдать осторожность. Если в команде указано имя X, которое описано до этой команды (как в нашем примере), тогда, встретив ее, ассемблер уже будет знать, что X обозначает двойное слово, и потому правильно поймет, что в данной команде имеется в виду дальний косвенный переход. Но если имя X будет описано позже, тогда ассемблер, встретив команду перехода, не будет знать, какого вида переход имеется в виду в данном случае – то ли близкий переход по метке, то ли близкий косвенный переход, то ли дальний косвенный переход. В такой ситуации, как уже говорилось в разд. 4.1, ассемблер предполагает, что указанное имя – это метка из текущего сегмента команд, и формирует машинную команду близкого перехода. Затем же, когда будет обнаружено, что в текущем сегменте такой метки нет, ассемблер зафиксирует ошибку. Чтобы ее избежать, надо в случае ссылки вперед при дальнем косвенном переходе явно указать, что имя обозначает переменную размером в двойное слово. Для этого используется оператор PTR:

```
JMP DWORD PTR X
```

Как уже отмечалось, подобного рода уточнения для ссылок вперед приходится делать и при близких переходах. Если собрать вместе все, что уже говорилось про ссылки вперед в командах безусловного перехода, то получится следующее правило: если X – ссылка вперед, тогда команду безусловного перехода следует записывать так:

```
JMP X             – близкий прямой длинный
JMP SHORT X       – близкий прямой короткий
JMP FAR PTR X     – дальний прямой
JMP WORD PTR X    – близкий косвенный
JMP DWORD PTR X   – дальний косвенный
```

Аналогичного рода уточнения надо делать и тогда, когда в команде JMP указана косвенная ссылка, например JMP [BX]; по умолчанию ассемблер рассматривает подобного рода команду как близкий косвенный переход.

Что же касается случая, когда X – ссылка назад, то вид перехода обязательно надо уточнять только в одном случае – при дальнем прямом переходе: ассемблер, даже зная, что X – метка из другого сегмента команд, все равно не будет рассматривать команду JMP X как дальний переход. Дело в том, что, встречая любую метку, ассемблер автоматически приписывает ей тип NEAR (близкий) и далее руководствуется этим типом. (Впрочем в ЯА есть директива LABEL, с помощью которой метке можно приписать и тип FAR – см. разд. 14.6.) Изменить же этот тип можно (в рамках одной команды) только с помощью оператора PTR.

Отметим, что NEAR и FAR – это имена стандартных констант со значениями соответственно -1 (0FFFFh) и -2 (0FFFEh). Именно эти значения выдает оператор TYPE, если в качестве его операнда указать не имя переменной, а метку (или имя процедуры); например: TYPE L = NEAR.

9.2. Подпрограммы-процедуры

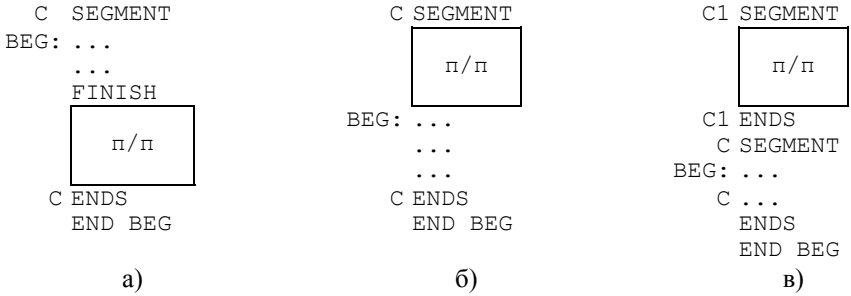
Довольно часто в программах, особенно больших, приходится несколько раз решать одну и ту же подзадачу и потому приходится многократно выписывать одинаковую группу команд, решающих эту подзадачу. Чтобы избежать повторного выписывания такой группы команд, ее обычно выписывают один раз и оформляют соответствующим образом, а затем в нужных местах программы просто передают управление на эти команды, которые, проработав, возвращают управление обратно. Такая группа команд, которая решает некоторую подзадачу и которая организована таким образом, что ее можно использовать любое число раз и из любых мест, называется подпрограммой. По отношению к подпрограмме остальную часть программы принято называть основной программой.

Легко понять, что подпрограммы – это аналог процедур из языков высокого уровня: сама подпрограмма – это аналог описания процедуры, а обращение к подпрограмме – аналог оператора процедуры. Однако реализация подпрограмм в ЯА – вещь более сложная, чем описание процедур в языках

высокого уровня. Дело в том, что в этих языках за нас многое делают трансляторы, а при программировании на ЯА нам никто не помогает, поэтому здесь все приходится делать самим. Какие проблемы возникают при использовании подпрограмм и как они решаются, мы и рассмотрим далее.

9.2.1. Где размещать подпрограмму?

Первая проблема – где размещать подпрограмму? Вообще говоря, где угодно. Но при этом надо понимать, что сама по себе подпрограмма не должна выполняться, а может выполняться лишь тогда, когда к ней обратятся. Поэтому размещать ее надо так, чтобы на нее случайно не попадало управление. Если имеется несколько подпрограмм, то их обычно размещают рядом. Обычно подпрограммы (п/п) размещают либо в конце сегмента команд за командой FINISH (см. рис. а), либо в самом начале этого сегмента – перед той командой, с которой должно начинаться выполнение программы (см. рис. б). В больших программах подпрограммы нередко размещают в отдельном сегменте команд (см. рис. в).



а) б) в)

9.2.2. Как оформлять подпрограмму?

Вторая проблема – как описывать подпрограмму? В общем случае группу команд, образующих подпрограмму, можно никак не выделять в тексте программы. Однако в ЯА подпрограммы принято оформлять специальным образом – в виде процедур. Это дает определенную выгоду, о которой будет сказано чуть позже. В дальнейшем мы будем использовать только такие подпрограммы, которые описаны в виде процедур.

Описание подпрограммы в виде процедуры выглядит так:

```
<имя процедуры> PROC <параметр>
                <тело процедуры>
<имя процедуры> ENDP
```

Как видно, перед телом процедуры (ее командами) ставится директива PROC (procedure), а за ним – директива ENDP (end of procedure). В обеих этих директивах указывается одно и то же имя – имя, которое мы дали подпрограмме, процедуре. Следует обратить внимание, что в директиве PROC после имени не ставится двоеточие (как и в других директивах), однако это имя считается меткой, считается, что оно метит первую команду процедуры.

Например, имя процедуры можно указать в команде перехода, и тогда будет осуществлен переход на первую команду процедуры.

У директивы PROC есть параметр – это либо NEAR (близкий), либо FAR (дальний). Параметр может и отсутствовать, тогда считается, что он равен NEAR (в связи с этим параметр NEAR обычно не указывается). При параметре NEAR или при отсутствии параметра процедура называется "близкой", при параметре FAR – "дальней". К близкой процедуре можно обращаться только из того сегмента команд, где она описана, и нельзя обращаться из других сегментов, а к дальней процедуре можно обращаться из любых сегментов команд (в том числе и из того, где она описана). В этом и только в этом различие между близкими и дальними процедурами.

Отметим, что в ЯА имена и метки, описанные в процедуре, не локализуются внутри нее, поэтому они должны быть уникальными, не должны совпадать с другими именами, используемыми в программе. В частности, хотя в ЯА и можно описать одну процедуру внутри другой, но никакой выгоды это не дает, поэтому в ЯА обычно не используется вложенность процедур.

9.2.3. Вызов процедур и возврат из них

Следующая проблема – как осуществляются вызовы процедур и возвраты из них? При программировании на языке высокого уровня для запуска процедуры в работу достаточно лишь выписать ее имя и ее фактические параметры. При этом нас не волнует, как процедура начинает работать, как она возвращает управление в основную программу: за все это отвечает транслятор. Но если мы программируем на ЯА, тогда все переходы между основной программой и процедурой приходится реализовывать нам самим. Рассмотрим, как это делается.

Здесь две проблемы: как из основной программы заставить работать процедуру и как вернуться из процедуры в основную программу. Первая проблема решается просто: достаточно выполнить команду перехода на первую команду процедуры, т. е. указать в команде перехода имя процедуры. Сложнее со второй проблемой. Дело в том, что обращаться к процедуре можно из разных мест основной программы, а потому и возвращаться из процедуры надо в разные места. Сама процедура, конечно, не знает, куда ей надо вернуть управление, зато это знает основная программа. Поэтому при обращении к процедуре основная программа обязана сообщить ей так называемый адрес возврата – адрес той команды основной программы, на которую процедура обязана сделать переход по окончании своей работы. Обычно это адрес команды, следующей за командой обращения к процедуре. Именно этот адрес основная программа и сообщает процедуре, именно по нему процедура и выполняет возврат в основную программу. Поскольку при разных обращениях к процедуре ей указывают разные адреса возврата, то она и возвращает управление в разные места основной программы.

Как сообщать адрес возврата? Это можно сделать по-разному. Во-первых, его можно передать через регистр: основная программа записывает в некоторый регистр адрес возврата, а процедура извлекает его оттуда и делает по нему переход. Во-вторых, это можно сделать через стек: прежде чем обратиться к процедуре, основная программа записывает адрес возврата в стек, а процедура затем считывает его отсюда и использует для перехода. В ПК принято передавать адрес возврата через стек, поэтому в дальнейшем мы будем рассматривать только этот способ передачи адреса возврата.

Передачу адреса возврата через стек и возврат по этому адресу можно реализовать с помощью тех команд, которые мы уже знаем. Однако в реальных программах процедуры используются очень часто, поэтому в систему команд ПК включены специальные команды, которые упрощают реализацию переходов между основной программой и процедурами. Это команды CALL и RET. Основные варианты этих команд следующие:

Вызов процедуры (переход с возвратом): `CALL <имя процедуры>`
Возврат из процедуры (return): `RET`

Команда CALL записывает адрес следующей за ней команды в стек и затем осуществляет переход на первую команду указанной процедуры. Команда RET считывает из вершины стека адрес и выполняет переход по нему.

Рассмотрим следующий пример. Пусть мы отлаживаем свою программу и для этого вставляем в разные ее места отладочную печать одних и тех же переменных X и Y. Такая печать осуществляется несколькими командами, они многократно повторяются, поэтому имеет смысл описать эту печать как процедуру и указывать в основной программе лишь короткие обращения к ней:

```
;основная программа  ;процедура
    ...              PR PROC
    CALL PR             OUTINT X
(a) ...                 OUTINT Y,8
    ...                 NEWLINE
    CALL PR             RET
(b) ...              PR ENDP
```

Когда выполняется первая команда CALL, то она записывает в стек адрес следующей за ней команды (адрес a) и передает управление на начало процедуры PR – на команду OUTINT X. Начинает работать процедура: она печатает X и Y и переводит строку. После этого команда RET извлекает из стека находящийся там адрес a и делает переход по нему. Тем самым возобновляется работа основной программы – с команды, следующей за первой командой CALL. Повторное обращение к процедуре происходит аналогично, но по второй команде CALL в стек уже будет записан другой адрес – адрес b, поэтому процедура на этот раз вернет управление в другое место основной программы – на команду, следующую за второй командой CALL.

Теперь кое-что уточним относительно команд CALL и RET.

Напомним, что в ПК адрес команды, которая должна быть выполнена следующей, задается регистрами CS и IP. Если описание процедуры размещено в том же сегменте команд, где мы обращаемся к ней, тогда переход на нее и возврат из нее должны быть близкими, т. е. должен меняться только указатель команд IP и не должен меняться сегментный регистр CS, т. к. мы все время остаемся в одном и том же сегменте команд. Но если процедура расположена в другом сегменте команд, тогда переход на нее и возврат из нее должны быть дальними, т. е. должны меняться оба этих регистра.

Все это учитывается, и на самом деле в ПК имеется по два варианта машинных команд CALL и RET. Если через AB обозначить адрес возврата – адрес (смещение) команды, следующей за командой CALL, тогда действия обоих вариантов команды CALL P, где P – имя процедуры, можно описать так:

```
близкий вызов: AB->стек, IP:=offset P
дальний вызов: CS->стек, AB->стек, CS:=seg P, IP:=offset P
```

Действия же двух вариантов команды RET таковы:

Близкий возврат: `стек->IP`
Дальний возврат: `стек->IP, стек->CS`

Ясно, что команды CALL и RET должны действовать согласованно: при близком вызове процедуры и возврат из нее должен быть близким, а при дальнем вызове и возврат должен быть дальним, иначе программа будет работать неправильно. В то же время в ЯА оба варианта каждой из этих команд записываются одинаково. Естественно возникает вопрос: как же тогда в ЯА указываются типы переходов между основной программой и процедурой? Ответ таков: явно это не указывается, эту проблему решает за нас ассемблер. Если мы описали процедуру как близкую, тогда все команды CALL, в которых указано имя этой процедуры, ассемблер будет транслировать в машинные команды близкого вызова, а все команды RET, расположенные внутри этой процедуры, ассемблер будет транслировать в машинные команды близкого возврата. Если же процедура описана как дальняя, тогда все обращения к ней будут транслироваться как дальние вызовы, а все команды RET внутри нее – как дальние возвраты. (Вне процедур RET рассматривается как близкий возврат.) Таким образом, ассемблер берет на себя обязанность следить за соответствием между командами CALL и RET, и у нас об этом не должна болеть голова. Именно в этом выгода от описания подпрограмм в виде процедур, именно из-за этого в ЯА и принято описывать подпрограммы как процедуры.

Однако здесь есть одна тонкость. Ассемблер выберет правильный вариант машинной команды для CALL, только если процедура была описана раньше этой команды. Если же процедура будет описываться позже, то ассемблер, встретив команду CALL и еще не зная тип этой процедуры (близкая она или дальняя), предполагает, что она близкая (а так чаще всего и бывает), и потому всегда в подобной ситуации формирует машинную команду близкого вы-

зова. Но если потом ассемблер обнаружит, что данная процедура описана как дальняя, то он зафиксирует ошибку. Чтобы не было такой ошибки, при обращении к дальней процедуре, которая будет описываться позже, надо в команде CALL с помощью оператора PTR явно указать, что процедура дальняя:

```
CALL FAR PTR P
```

9.2.4. Другие варианты команды CALL

Мы рассмотрели основной вариант команды CALL, когда в качестве ее операнда указывается имя процедуры. Но возможны и другие варианты операнда – точно такие же, как в команде безусловного перехода JMP, за исключением случая с оператором SHORT (считается, что процедуры не располагаются рядом с командами их вызова, и потому в ПК не предусмотрен короткий переход с возвратом). Примеры:

```
NA DW P
FA DD Q
   ...
 P PROC
   ...
P1: ...
   ...
 P ENDP
 Q PROC FAR
   ...
Q1: ...
   ...
 Q ENDP
   ...
   CALL P1               ;близкий переход на P1 с возвратом
   CALL FAR PTR Q1       ;дальний переход на Q1 с возвратом
   CALL Q1               ;близкий (!) переход на Q1 с возвратом
   CALL NA               ;близкий вызов процедуры P
   CALL FA               ;дальний вызов процедуры Q
   LEA BX,Q
   CALL [BX]             ;близкий (!) вызов процедуры Q
   CALL DWORD PTR [BX]   ;дальний вызов процедуры Q
   ...
```

При использовании этих вариантов команды CALL надо соблюдать осторожность. Во-первых, надо следить за согласованностью типа перехода с возвратом с типом возврата по команде RET, т. к. в этих случаях ассемблер уже не отвечает на такую согласованность. Во-вторых, как и в команде JMP, при использовании в команде CALL ссылок вперед или косвенных ссылок следует, если надо, уточнять тип этих ссылок (по умолчанию при ссылке вперед ассемблер транслирует близкий прямой вызов, а при косвенной ссылке – близкий косвенный вызов).

9.3. Передача параметров через регистры

Теперь рассмотрим проблемы, связанные с параметрами процедур.

В языках высокого уровня для того, чтобы задать фактические параметры для процедуры, достаточно лишь выписать их в операторе вызова процедуры.

В ЯА проблема задания параметров решается не столь просто, поэтому мы ее рассмотрим подробно.

Попутно рассмотрим и то, как возвращается результат процедуры. Отметим, что в ЯА нет формального деления на процедуры и функции, то и другое называется процедурами. Но содержательно их, конечно, можно разделить на функции и "чистые" процедуры в зависимости от того, вырабатывают они результат или нет.

Передавать фактические параметры процедуре можно по-разному. Простейший способ – передавать параметры через регистры: основная программа записывает фактические параметры в какие-то регистры, а процедура затем берет их оттуда и использует в своей работе. Аналогично можно поступить и с результатом, если он имеется: процедура записывает свой результат в какой-то регистр, а основная программа затем извлекает его оттуда. Через какие регистры передавать параметры и результат? Это личное дело автора программы, он сам определяет эти регистры.

9.3.1. Передача параметров по значению

Рассмотрим такой пример. Пусть надо вычислить c=max(a,b)+max(5,a-1), где все числа – знаковые и размером в слово. Вычисление max(x,y) опишем как процедуру-функцию, при этом договоримся о следующем: первый параметр (x) основная программа должна передавать через регистр AX, второй параметр (y) – через регистр BX, а результат (max) процедура должна возвращать через регистр AX. При этих условиях процедура и соответствующий фрагмент основной программы выглядят так (для примера опишем процедуру как дальнюю):

```
;процедура: AX=max(AX,BX)
  MAX PROC  FAR
      CMP AX,BX
      JGE MAX1
      MOV AX,BX
MAX1: RET
  MAX ENDP
```

```
;основная программа
...
MOV AX,A  ;AX:=a
MOV BX,B  ;BX:=b
CALL MAX  ;AX:=max(a,b)
MOV C,AX  ;спасти AX
MOV AX,5  ;AX:=5
MOV BX,A
DEC BX    ;BX:=a-1
CALL MAX  ;AX:=max(5,a-1)
ADD C,AX  ;C:=max(a,b)+max(5,a-1)
...
```

Если воспользоваться терминологией языка Паскаль, то в этом примере параметры передаются по значению: перед обращением к процедуре основная процедура вычисляет значения фактических параметров и именно эти значения записывает в регистры. Теперь же рассмотрим другой способ передачи параметров – по ссылке.

9.3.2. Передача параметров по ссылке

Возьмем следующую процедуру на языке Паскаль:

```
procedure D(var x:integer); begin x:=x div 16 end;
```

Пусть в программе есть такие обращения к ней: D(A) и D(B), где A и B – имена переменных, значениями которых являются неотрицательные числа.

Как видно, процедура что-то присваивает своему параметру. В терминах машинного языка присваивание означает запись в какую-то ячейку памяти, а чтобы записать что-то в ячейку, надо знать адрес (имя) этой ячейки. Поэтому процедуре надо знать адрес той ячейки (A или B), в которую она должна сделать запись, и этот адрес обязана ей сообщить основная программа. Таким образом, передача параметра по ссылке означает передачу адреса (имени) ячейки, соответствующей фактическому параметру.

Как передавать адрес? Через регистр: основная программа записывает в какой-то регистр адрес фактической переменной, а процедура берет его оттуда. Какой это регистр? Вообще говоря, любой, но лучше, если это будет регистр-модификатор, т. е. BX, BP, SI или DI, т. к. процедуре придется модифицировать по этому регистру.

Пусть для нашей процедуры D мы выбрали регистр BX. Это означает, что к началу ее выполнения в регистре BX будет находиться адрес той ячейки (A или B), содержимое которой она обязана изменить. В подобной ситуации добраться до этой ячейки, как мы уже знаем (это пример на косвенную ссылку), можно с помощью конструкции [BX].

С учетом всего сказанного получаем следующий фрагмент основной программы, соответствующий обращениям D(A) и D(B), и следующую процедуру D (на строки с командами PUSH и POP пока не обращать внимания):

```
;основная программа     ;процедура: BX=адрес x, x:=x div 16
...                     D PROC
LEA BX,A  ;BX=адрес A     PUSH CX               ;спасти CX
CALL D    ; D(A)          MOV CL,4
LEA BX,B  ;BX=адрес B     SHR WORD PTR [BX],CL   ;x:=x div 16
CALL D    ; D(B)          POP CX             ;восстановить CX
...                       RET
                        D ENDP
```

9.3.3. Сохранение регистров в процедуре

Как видно, нашей процедуре D потребовался регистр CL, в который она заносит величину сдвига. Возникает вопрос: имеет ли право процедура менять, портить этот регистр?

Это очень важная проблема. Дело в том, что в ПК не так уж и много регистров и в то же время чуть ли не в каждой команде используется тот или иной регистр. Поэтому с большой вероятностью основной программе и процедуре могут потребоваться для работы одни и те же регистры, и тем самым они будут "мешать" друг другу. Конечно, можно договориться, чтобы основная программа и процедура пользовались разными регистрами, однако сделать это очень сложно – уж очень мало регистров в ПК. Поэтому обычно поступают иначе: разрешают и основной программе, и процедуре пользоваться одними и теми же регистрами, но при этом требуют от процедуры, чтобы она сохраняла те значения

регистров, которые в них записала основная программа. Достичь этого просто: в начале своей работы процедура должна спасти в стеке значения тех регистров, которые ей потребуются для работы, после чего она может использовать эти регистры как угодно, а перед выходом она должна восстановить прежние значения этих регистров, считав их из стека. Именно так мы и поступили в процедуре D. Правда, здесь есть одна тонкость: нам надо сохранить значение байтового регистра CL, но, как мы знаем, в стек можно записывать только слова. Поэтому спасаем в стеке не регистр CL, а весь регистр CX и в конце также восстанавливаем весь регистр CX.

Такое сохранение регистров настоятельно рекомендуется делать в любой процедуре, даже если явно видно, что основная программа не пользуется теми же регистрами, что и процедура. Дело в том, что исходный текст программы в дальнейшем может измениться (а это происходит очень часто), и может оказаться так, что после этих изменений основной программе потребуются эти регистры. И хорошо, если мы при этом вспомним, что надо подправить процедуру. Чаще же всего про это забывают, и потому основная программа и процедура начинают мешать друг другу. Поэтому лучше сразу предусмотреть в процедуре сохранение регистров, тогда при любых изменениях основная программа может не волноваться за свои регистры. (Для поддержки этого стиля программирования в систему команд ПК, начиная с процессора 80186, даже введены специальные команды PUSHA и POPA, которые позволяют сохранять в стеке и восстанавливать из стека значения регистров общего назначения – см. разд. 8.2.)

Отметим, что сохранять значение регистра, через который процедура возвращает результат, конечно, не надо, поскольку в изменении этого регистра и заключается цель работы процедуры.

9.3.4. Передача параметров сложных типов

Теперь рассмотрим еще один случай передачи параметра по ссылке – когда параметром является данное сложного типа (массив, структура и т. п.). Даже если процедура не меняет это данное, ей все равно обычно передается не само данное (для него может просто не хватить регистров), а его начальный адрес. Зная этот адрес, процедура может легко добраться до любой части этого данного.

Рассмотрим следующий пример. Пусть имеются массивы из чисел без знака

```
X DB 100 DUP(?)
Y DB 25 DUP(?)
```

и требуется записать в регистр DL сумму максимальных элементов этих массивов: DL=max(X[i])+max(Y[i]).

Поскольку здесь дважды приходится находить максимальный элемент массива, то, чтобы не повторяться, имеет смысл описать это действие в виде процедуры. Назовем ее MAX и опишем ее при условии, что начальный адрес массива передается процедуре через регистр BX, количество элементов в массиве – че-

рез регистр CX, а свой результат процедура возвращает через регистр AL. При этих соглашениях описание процедуры и фрагмент основной программы, решающий поставленную задачу, выглядят так:

```
;процедура MAX: AL=max(W[0..N-1]), где BX=нач.адрес W, CX=N
  MAX PROC
      PUSH BX   ;спасти регистры,
      PUSH CX   ; используемые в процедуре
      MOV AL,0  ;AL = 0 (нач.значение максимума)
MAX1: CMP [BX],AL
      JLE MAX2  ;W[i]>AL ==> AL:=W[i]
      MOV AL,[BX]
MAX2: INC BX
      LOOP MAX1
      POP CX    ;восстановить регистры
      POP BX
      RET       ;выход из процедуры
  MAX ENDP
      ...
;фрагмент основной программы для вычисления
DL=max(X[i])+max(Y[i])
      LEA BX,X  ;BX=нач.адрес массива X
      MOV CX,100   ;CX=число элементов в X
      CALL MAX  ;AL=max(X[i])
      MOV DL,AL ;спасти AL
      LEA BX,Y  ;BX=нач.адрес массива Y
      MOV CX,25 ;CX=число элементов в Y
      CALL MAX  ;AL=max(Y[i])
      ADD DL,AL
      ...
```

9.4. Передача параметров через стек

Передача параметров через регистры – вещь удобная и используется очень часто. Однако делать так можно, только если параметров немного; если же параметров много, то на них попросту не хватит регистров. В таком случае используют иной способ передачи параметров – через стек: основная программа записывает фактические параметры (их значения или адреса) в стек, а процедура затем их оттуда извлекает. Конкретно реализовать эту идею можно по-разному, мы рассмотрим тот способ передачи параметров через стек, который используется транслятором языка Турбо Паскаль.

Пусть процедура P имеет k параметров: P(a1,a2,...,ak). Договоримся, что перед обращением к процедуре основная программа записывает параметры в стек. В каком порядке? Это решает автор программы. Мы будем считать, что параметры записываются в стек слева направо: сначала записывается 1-й параметр, затем 2-й и т. д. Если для определенности договориться, что каждый параметр имеет размер слова и его не надо вычислять (эти ограничения легко обходятся), тогда команды основной программы, реализующие обращение к процедуре, будут следующими (справа показано состояние стека после выполнения этих команд, т. е. на входе в процедуру):

```
   ;обращение P(a1,a2...,ak)
     PUSH a1                          SP→  AB
     PUSH a2                               ak
     ...                                   ...
     PUSH ak                               a1
     CALL P                          (AB)  //////
(AB) ...
```

Теперь начинает работать процедура. И сразу возникает проблема: как ей добраться до параметров? Это проблема доступа к элементам стека без считывания их из стека. Как она решается, мы уже знаем: надо воспользоваться регистром BP, т. е. заслать в него адрес вершины стека (содержимое регистра SP), а затем использовать выражения вида [BP+i] для доступа к параметрам процедуры. Однако здесь есть одно "но": мы портим регистр BP, а он может использоваться в основной программе. Поэтому сначала надо спасти прежнее значение этого регистра и только затем уж пересылать в него значение регистра SP. Таким образом, выполнение процедуры должно начинаться со следующих команд, которые называют "входными" действиями процедуры (справа изображено состояние стека после этих команд; для определенности считаем, что процедура близкая, поэтому адрес возврата занимает одно слово в стеке):

```
;"входные" действия процедуры          SP,BP→  BPст
   P PROC                                  +2   AB
     PUSH BP   ;спасти BP                  +4   ak
     MOV BP,SP ;BP - на вершину стека           ...
     ...                                        a1
     команды процедуры                          //////
```

Как видно из рисунка, после записи в стек старого значения BP (BPст) для доступа к параметрам процедуры надо использовать следующие выражения: [BP+4] – для доступа к последнему параметру, [BP+6] – для доступа к предпоследнему параметру и т. д. Например, считывание последнего параметра в регистр AX (AX:=ak) осуществляется командой MOV AX,[BP+4].

Далее идут команды собственно процедуры. После их завершения процедура обязана выполнить действия, которые называются "выходными". Рассмотрим их. Прежде всего отметим, что к этому моменту стек должен быть в том же состоянии, в каком он был после "входных" действий. (Если это не так, то восстановить данное состояние можно командой MOV SP,BP.) Тогда к концу работы процедуры в вершине стека будет находиться старое значение регистра BP. Считываем его и восстанавливаем BP. Теперь в вершине стека окажется адрес возврата. Казалось бы, можно выполнить команду RET и уйти из процедуры, однако это не так – надо еще очистить стек от параметров, которые уже не нужны. Кто должен делать эту очистку – процедура или основная программа? Конечно, это может сделать основная программа, для чего в ней после команды CALL P надо выполнить команду ADD SP,2*k.

Однако лучше, если очистку стека будет делать сама процедура. Дело в том, что обращений к процедуре много, поэтому в основной программе команду ADD придется выписывать многократно, а процедура – одна, поэтому в ней такую команду надо выписать только раз. Это общее правило: если что-то может сделать и основная программа, и процедура, то лучше, если это "что-то" будет делать процедура. Так меньше команд получается.

Итак, процедура должна сначала очистить стек от параметров и только затем передать управление по адресу возврата. Чтобы упростить реализацию этой пары действий, в систему команд ПК введен расширенный вариант команды RET – с непосредственным операндом, который трактуется как число без знака:

```
RET i16
```

По этой команде сначала из стека удаляется адрес возврата, затем стек очищается на указанное операндом число байтов и далее выполняется переход по адресу возврата:

```
стек -> IP, [стек -> CS,] SP:=SP+i16
```

(действие "стек -> CS" выполняется лишь при дальнем возврате).

Сделаем несколько замечаний. Во-первых, команда RET – это на самом деле команда RET 0, т. е. возврат без очистки стека. Во-вторых, операнд команды указывает, на сколько байтов, а не слов надо очищать стек, поэтому для очистки стека от k параметров, каждый из которых имеет размер слова, надо указывать операнд 2*k, а не k. В-третьих, в операнде не должен учитываться адрес воз-врата – команда RET считывает его до очистки стека.

С учетом всего сказанного "выходные" действия процедуры таковы:

```
;"выходные" действия процедуры
  POP BP   ;восстановить старое значение BP
  RET 2*k  ;очистка стека от k параметров-слов и возврат
P ENDP
```

После такого возврата из процедуры состояние стека будет тем же самым, каким оно было до выполнения команд обращения к процедуре, т. е. до выполнения команд записи параметров в стек. Тем самым в стеке уничтожаются все следы обращения к процедуре, а это как раз то, что нам и надо.

Такова общая схема передачи параметров через стек. Еще раз напомним, что этот способ передачи параметров универсален, его можно использовать при любом числе параметров. Однако этот способ более сложный, чем передача параметров через регистры, поэтому, если можно, следует передавать параметры через регистры, так будет проще и короче. Что же касается результата процедуры, то он крайне редко передается через стек и обычно передается через регистр.

В качестве конкретного примера опишем процедуру NULL(A,N), которая обнуляет N байт памяти (в сегменте данных) начиная с адреса A, при условии, что адрес первого параметра (A) и значение второго параметра (N) передаются через

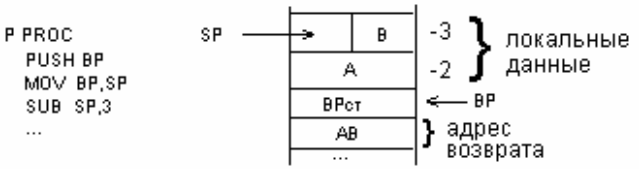
стек. Описание процедуры приведено справа, а слева показана реализация обращения NULL(X,100), по которому обнуляется 100-байтовый массив X:

```
X DB 100 DUP(?)          NULL PROC
  ...                           PUSH BP     ;"входные" действия
;вызов NULL(X,100)              MOV BP,SP
  LEA AX,X                      PUSH BX     ;спасти BX и CX
  PUSH AX   ;адр.X->стек        PUSH CX     ; ("над" BPст)
  MOV AX,100                    MOV CX,[BP+4]          ;CX=A
  PUSH AX   ;100->стек          MOV BX,[BP+6]          ;BX=N
  CALL NULL               NULL1: MOV BYTE PTR [BX],0   ;обнуление
  ...                           INC BX      ; A[0..N-1]
                                LOOP NULL1
                                POP CX      ;восстан. CX и BX
                                POP BX
                                POP BP      ;"выходные" действия
                                RET 4
                          NULL ENDP
```

9.5. Локальные данные процедур

Во многих процедурах не возникает проблемы с хранением локальных данных (величин, нужных только на время выполнения процедуры) – для них достаточно и регистров. Однако если в процедуре много локальных данных, тогда возникает вопрос: где отводить место для них? Конечно, можно выделить место в сегменте данных, но это плохо, т. к. большую часть времени это место памяти будет пропадать зря. Лучше выделять это место в стеке: при входе в процедуру в вершине стека "захватывается" нужное число байтов для локальных данных, а перед выходом это место освобождается. В таком случае место в памяти занимается только на время выполнения процедуры.

Такой "захват" места в стеке можно делать как в случае передачи параметров процедуре через регистры, так и в случае передачи параметров через стек. Для этого надо запомнить в стеке текущее значение регистра BP и затем установить его на вершину стека (при передаче параметров через стек это есть ничто иное, как "входные" действия), после чего надо уменьшить значение указателя стека SP на число "захватываемых" байтов. Например, если некоторой процедуре P требуется 3 байта (скажем, 2 байта под локальную переменную A и 1 байт под локальную переменную B), тогда указанные действия реализуются следующими командами (справа изображено состояние стека после этих команд):



После этого доступ к локальным данным осуществляется с помощью выражений вида [BP-k]; например, [BP-2] – это адрес локальной переменной A, а [BP-3] – адрес локальной переменной B.

При завершении работы процедуры надо выполнить такие действия:

```
  ...
  MOV SP,BP  ;отказ от места для локальных данных
  POP BP     ;восстановление старого значения BP
  RET        ;(или RET n) - возврат из процедуры
P ENDS
```

В качестве конкретного примера опишем близкую процедуру DIF, которая подсчитывает, сколько различных символов входит в заданную строку (символьный массив), при условии, что начальный адрес строки передается через регистр BX, а длина строки – через CX, а результат возвращается через AX.

Воспользуемся следующим алгоритмом. Заводим в стеке локальный массив из 256 байт – по одному на каждый возможный символ, причем символу с кодом k поставим в соответствие байт стека с адресом [BP-256+k], т. е. нумеруем элементы этого стекового массива "сверху вниз", и обнуляем этот массив. Затем просматриваем заданную строку и для каждого входящего в нее символа записываем 1 в соответствующий элемент локального массива; тем самым в этом массиве будут отмечены все символы, которые хотя бы раз входили в строку. В конце подсчитываем число единиц в локальном массиве.

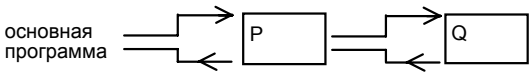
```
  DIF PROC
;"входные" действия
      PUSH BP
      MOV BP,SP
      SUB SP,256    ;занять место в стеке под лок. массив
      PUSH BX       ;спасти регистры, используемые в процедуре
      PUSH CX
      PUSH SI
;обнуление локального массива
      MOV AX,CX          ;сохранить длину строки
      MOV CX,256         ;длина локального массива
      MOV SI,0           ;индекс в этом массиве (от 0 до 255)
DIF1: MOV BYTE PTR [BP-256+SI],0
      INC SI
      LOOP DIF1
      MOV CX,AX               ;восстановить в CX длину строки
;просмотр строки и запись 1 в локальный массив
      MOV AH,0                ;для расширения AL --> AX
DIF2: MOV AL,[BX]             ;код очередного символа строки
      MOV SI,AX               ;перепись его в модификатор SI
      MOV BYTE PTR [BP-256+SI],1 ;запись 1 в локальный массив
      INC BX                  ;адрес следующего символа
      LOOP DIF2
;подсчет числа 1 в локальном массиве
      MOV AX,0           ;счетчик единиц
      MOV CX,256         ;длина локального массива
```

```
      MOV SI,0           ;индекс в этом массиве
DIF3: CMP BYTE PTR [BP-256+SI],1
      JNE DIF4
      INC AX
DIF4: INC SI
      LOOP DIF3
;"выходные" действия
      POP SI             ;восстановить регистры
      POP CX
      POP BX
      MOV SP,BP          ;освободить стек от лок. массива
      POP BP             ;восстановить старое значение BP
      RET
  DIF ENDP
```

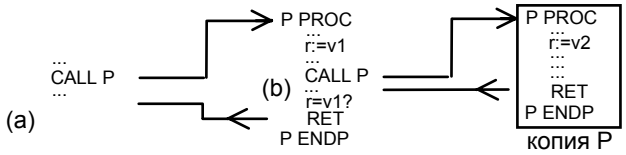
9.6. Рекурсивные процедуры

Напомним, что процедура называется рекурсивной, если она обращается сама к себе непосредственно (прямая рекурсия) или через другие процедуры (косвенная рекурсия). Мы не будем объяснять, как следует описывать рекурсивные процедуры (это не тема данной книги), а будем считать, что уже имеется описание рекурсивной процедуры и наша задача – рассмотреть, как это описание реализуется на ЯА. Учитывая, что многие программисты боятся рекурсии, сразу отметим, что ничего сложного в этой реализации нет, надо лишь придерживаться определенных правил, о которых здесь и будет рассказано.

В общем случае одна процедура может обратиться к другой процедуре. Например, основная программа может вызвать процедуру P, которая в свою очередь вызывает процедуру Q:



Как частный случай, вызываемой процедурой Q может быть сама вызывающая процедура P (Q=P), и тогда P окажется рекурсивной процедурой. Таким образом, в рекурсивной процедуре есть команда вызова ее самой, т. е. передача управления на ее начало (мы ограничимся рассмотрением только случая прямой рекурсии):



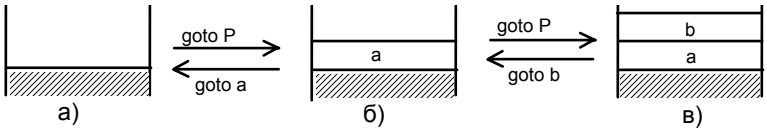
(На строчки с r пока не обращать внимание.)

Отметим, что рекурсивное (повторное) обращение лучше рассматривать не как вызов самой процедуры, а как вызов ее копии: можно считать, что

имеется не один экземпляр процедуры, а много таких экземпляров, и при рекурсивном обращении мы из одного экземпляра вызываем другой экземпляр. Такая трактовка рекурсивных вызовов более наглядна и понятна, ее мы и будем придерживаться в дальнейшем.

Поскольку рекурсивная процедура вызывает саму себя (свою копию), естественно возникает вопрос: а не зациклится ли она, выйдет ли она когда-нибудь из рекурсии? Ответ: не зациклится, если она описана правильно, если в ней есть нерекурсивная ветвь, т. е. такой путь вычисления, на котором рекурсивный вызов обходится, и если при каком-то обращении вычисление пойдет по этой ветви.

Пусть, к примеру, при первом (из основной программы) вызове процедуры P внутренняя команда CALL сработает, а при втором (рекурсивном) вызове она уже обходится. Тогда происходит следующее: основная программа вызывает первую копию процедуры P и заносит в стек адрес возврата a (см. рис. б). В этой копии выполняется команда CALL P, по которой в стек заносится адрес возврата b (см. рис. в) и передается управление на начало второй копии процедуры. Согласно нашему предположению, команда CALL в этой копии не выполняется, поэтому вычисление доходит до команды RET, которая считывает из стека адрес возврата b и передает по нему управление (см. рис. б). Тем самым происходит возврат в первую копию процедуры, и ее работа возобновляется. Команда RET из этой копии считывает из стека адрес возврата a (см. рис. а) и передает по нему управление, в результате чего происходит возврат в основную программу. Зацикливания нет.



Из этого примера видно, что для правильной организации рекурсии очень важно то, что для передачи адресов возврата используется стек. Во-первых, благодаря стеку новый адрес возврата не "забивает" прежний адрес возврата, что было бы, если бы адреса передавались через регистр. Во-вторых, благодаря стеку адреса возврата извлекаются в нужном порядке – первым всегда берется адрес, записанный в стек последним. Важную роль стека для реализации рекурсивных процедур прекрасно понимали создатели ПК, поэтому они и ввели в систему команд ПК команды CALL и RET, которые организуют передачу адресов возврата именно через стек.

Теперь рассмотрим проблему, связанную с использованием регистров в рекурсивных процедурах. Предположим, что рекурсивная процедура P использует некоторый регистр r, не заботясь о сохранении его значения, и пусть она сначала присваивает этому регистру некоторую величину v1, а затем использует это значение, и пусть между этими действиями процедура обращается сама к себе (см. выше). Поскольку при рекурсивном вызове на-

чинают работать те же самые команды, то регистру r снова будет что-то присвоено, но уже, вообще говоря, какое-то новое значение v2. Почему не то же самое значение v1? Вспомним: и в циклах выполняются одни и те же команды, но на каждом шаге они оперируют с разными данными. Так и здесь: команда присваивания регистру r – та же самая, но присваиваемое значение может быть другим.

Итак, во второй копии нашей процедуры P регистру r присваивается новое значение v2. Пусть в этой копии нет рекурсивного вызова, тогда она завершает работу и возвращает управление в первую копию процедуры. Но какое значение теперь будет у регистра r: v1 или v2? Ясно, что v2, и это, конечно, нарушит работу первой копии процедуры, т. к. она, скорее всего, рассчитывала, что в r сохранится значение v1. Почему так произошло? А потому, что наша процедура не сохраняет значение регистра r. Если бы процедура на входе сохраняла значение регистра r, а на выходе восстанавливала его, тогда при возврате из второй копии процедуры в регистре r сохранилось бы нужное значение v1 и первая копия правильно бы продолжила свою работу.

Таким образом, если нерекурсивная процедура, вообще говоря, может сохранять, а может и не сохранять значения регистров, которые она использует, то рекурсивная процедура просто обязана сохранять значения всех регистров, которыми она пользуется. Причем для сохранения значений регистров должен использоваться стек, чтобы новая копия процедуры сохраняла значения регистров в новом месте.

Аналогичная проблема возникает, если процедура пользуется какими-то ячейками памяти для временного хранения своих промежуточных результатов. В этом легко убедиться, если в нашем примере рассматривать r не как регистр, а как ячейку. Поэтому, если рекурсивная процедура пользуется ячейками памяти, то она обязана на входе спасать прежние значения этих ячеек, а на выходе восстанавливать эти прежние значения. А где спасать? В стеке. Но стек – это тоже ячейки памяти. Поэтому, чтобы не было лишних инстанций, рекурсивной процедуре лучше вообще отказаться от хранения своих данных в обычных ячейках памяти и запоминать все нужное сразу в стеке.

Таковы требования, которых надо придерживаться при описании рекурсивных процедур. Если их соблюдать, тогда проблем с реализацией рекурсивных процедур не будет.

И последнее замечание. Передавать параметры и результат рекурсивной процедуры можно и через регистры, и через стек. Это не влияет на реализацию рекурсии.

В качестве конкретного примера рекурсивной процедуры опишем функцию F(n), вычисляющую n-е число Фибоначчи (n>=0) по следующему правилу:

$$F(n)=\begin{cases} 1, & n=0 \text{ или } n=1 \\ F(n-1)+F(n-2), & n>=2 \end{cases}$$

Договоримся, что аргумент функции передается через регистр AL, а значение функции возвращается через регистр BX. При этих условиях и при соблюдении требований к рекурсивным процедурам получаем такое описание функции:

```
;BX=F(n) – число Фибоначчи с номером n (AL=n)
  F PROC
    CMP AL,1    ;n>1 -> F1
    JA F1
;нерекурсивная ветвь
    MOV BX,1    ;n<=1 -> BX=F(n)=1
    RET
;рекурсивная ветвь
F1: PUSH AX     ;спасти AX (будем менять)
    DEC AL      ;AL=n-1
    CALL F      ;BX=F(n-1) (AX не изменится)
    PUSH BX     ;спасти F(n-1)
    DEC AL      ;AL=n-2
    CALL F      ;BX=F(n-2)
    POP AX      ;восстановить F(n-1), но уже в AX
    ADD BX,AX   ;BX=F(n)=F(n-2)+F(n-1)
    POP AX      ;восстановить исходное значение AX
    RET
  F ENDP
```

10. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

В данной главе рассматриваются способы представления и обработки строк переменной длины и списков – структур, размеры которых меняются в процессе выполнения программы.

10.1. Строковые команды. Префиксы повторения

Тех команд, которые мы уже знаем, вполне достаточно, чтобы запрограммировать любые операции над строками (последовательностями символов, байтов). Однако строки – столь важный тип данных, что во многие ЭВМ вводят специальные команды, упрощающие обработку строк. Особенность этих команд, называемых строковыми, в том, что одной или парой таких команд можно выполнить некоторую операцию сразу над всей строкой. Есть такие команды и в ПК, они и будут рассмотрены в этом разделе.

У строковых команд ПК много общего, поэтому, чтобы не повторяться, мы сначала подробно рассмотрим одну из них, а затем коротко опишем остальные.

10.1.1. Команда сравнения строк

Прежде всего отметим, что в строковых командах под "строкой" понимается не только последовательность байтов (символов), но и последовательность слов. В связи с этим каждая строковая операция представлена в ПК двумя командами: одна из них предназначена для обработки строк из байтов, а другая – для обработки строк из слов. В ЯА мнемокоды этих команд различаются тем, что в первом случае указывается буква B (byte), а в другом – буква W (word). Например, в командах сравнения строк (CMPS, compare strings) используются следующие мнемокоды:

```
CMPSB – сравнение строк из байтов.
CMPSW – сравнение строк из слов.
```

В целом действия этой пары команд совпадают, поэтому обычно про них говорят как про одну команду – команду CMPS, и только если надо, уточняют, какой именно вариант ее имеют в виду.

Команда CMPS записывается без операндов, ее действие можно описать так:

```
[DS:SI]=[ES:DI]? ; SI:=SI+d; DI:=DI+d,
```

где величина d определяется согласно следующей таблице (DF – флаг направления, direction flag):

	DF=0	*DF=1*
CMPSB	+1	-1
CMPSW	+2	-2

Прокомментируем действия команды CMPS.

У команды два операнда, но их местонахождение заранее известно, поэтому они явно и не указываются. CMPS принято называть командой сравнением строк, но на самом деле она сравнивает только пару элементов – элемент из одной строки с элементом из другой строки. При этом абсолютный адрес элемента из первой строки должен задаваться регистрами DS и SI, а абсолютный адрес элемента из второй строки – регистрами ES и DI.

Сравниваемые строки могут находиться в памяти далеко друг от друга, в разных сегментах памяти. Поэтому в общем случае строки нельзя сегментировать по одному сегментному регистру. В связи с этим в ПК договорились о том, что одна строка должна сегментироваться по регистру DS, т. е. DS должен указывать на начало сегмента памяти, в котором находится эта строка, а другая строка должна сегментироваться по регистру ES. Смещения же сравниваемых элементов (их адреса, отсчитанные от начала тех сегментов памяти, где расположены строки) договорились указывать в регистре SI (для элемента первой строки) и регистре DI (для элемента второй строки). Подчеркнем особо: в регистрах SI и DI указываются адреса элементов строк, а не их индексы, не их номера внутри строк.

Основное действие команды CMPS заключается в сравнении элемента одной строки с элементом другой строки. При этом команда CMPSB сравнивает байты, а команда CMPSW – слова. Это сравнение выполняется так же, как и в команде обычного сравнения CMP, т. е. путем вычитания операндов без записи куда-либо полученной разности. Главное здесь – формирование флагов, которые отражают результат сравнения и которые затем можно проверить командами условного перехода.

Но на этом действие команды не заканчивается. Она еще меняет значения регистров SI и DI и меняет так, чтобы в них оказались адреса соседних элементов строк. Но каких соседних? Это зависит от текущего значения флага направления DF: при DF=0 значения регистров увеличиваются, т. е. происходит переход вперед – к следующим элементам, а при DF=1 значения регистров уменьшаются, т. е. происходит переход назад – к предыдущим элементам. Разница же между командами CMPSB и CMPSW проявляется в том, что команда CMPSB меняет эти регистры на 1, а команда CMPSW – на 2, т. е. эти команды учитывают размеры элементов строк.

Как видно, действие команды зависит от значения флага направления DF. Сам по себе этот флаг не меняется, менять его должен автор программы. Сделать это можно с помощью следующих команд:

```
Очистка флага DF (clear DF): CLD
Установка флага DF (set DF): STD
```

По команде CLD флаг направления обнуляется (DF:=0), а по команде STD флагу присваивается 1 (DF:=1).

Установленное этими командами значение флага сохраняется до тех пор, пока не будет снова выполнена одна из этих команд. Отметим также, что

в самом начале выполнения программы значение флага DF может быть любым.

Итак, команда CMPS сравнивает пару элементов двух строк и автоматически настраивается на соседние элементы строк, обеспечивая продвижение по строкам в определенном направлении – вперед (от начала к концу) при DF=0 или назад при DF=0. Легко сообразить, что осталось только зациклить эту команду, чтобы можно было сравнить строки целиком.

Пусть N – это некоторая константа с положительным значением и пусть строка S1 из N байт расположена в сегменте памяти, на начало которого уже установлен сегментный регистр DS, а строка S2 из того же числа байтов находится в сегменте, на начало которого уже установлен регистр ES. Тогда проверить на равенство эти две строки можно с помощью следующих команд (слева строки просматриваются вперед, справа – назад):

```
; просмотр вперед                  ; просмотр назад
   CLD        ;DF=0 (вперед)         STD             ;DF=1 (назад)
   LEA SI,S1  ;DS:SI=начало S1       LEA SI,S1+N-1   ;DS:SI=конец S1
   LEA DI,S2  ;ES:DI=начало S2       LEA DI,S2+N-1   ;ES:DI=конец S2
   MOV CX,N   ;CX=длина строк        MOV CX,N        ;CX=длина строк
L: CMPSB   ;сравнить пару эл-тов  L: CMPSB      ;сравнить пару эл-тов
   JNE NOEQ   ;S1<>S2 --> NOEQ       JNE NOEQ        ;S1<>S2 --> NOEQ
   LOOP L     ;к следующей паре      LOOP L          ;к предыдущей паре
EQ: ...       ;S1=S2               EQ: ...           ;S1=S2
```

Как видно, в этих циклах нам не пришлось самим менять значения регистров SI и DI, это делает строковая команда. Собственно в этом и проявляется достоинство строковых команд, этим они и упрощают реализацию операций над строками. Однако создатели ПК на этом не успокоились и предоставили средство, упрощающее зацикливание строковых команд. Это префиксы повторения.

10.1.2. Префиксы повторения

В ПК имеется две команды без операндов, которые называются префиксами повторения. В ЯА каждая из них имеет несколько названий-синонимов:

1-й префикс:
```
REPE (repeat if equal; повторять, пока равно),
REPZ (повторять, пока ноль),
REP (повторять)
```

2-й префикс:
```
REPNE (repeat if not equal; повторять пока
                                ; не равно),
REPNZ (повторять, пока не ноль)
```

(Сравнение x=y можно представить как сравнение x-y=0, что и объясняет эквивалентность фраз "пока равно" и "пока ноль". О названии же просто REP будет сказано позже).

Префиксы повторения ставятся перед строковыми командами, причем в ЯА такой префикс обязательно должен записываться в одной строчке со строковой командой; например:

```
REPE CMPSB
```

Если поставить префикс повторения перед нестроковой командой, то он никак не будет сказываться, проработает как "пустая" команда.

Действие префикса повторения состоит в том, что он заставляет многократно повторяться следующую за ним строковую команду. При этом данная пара команд выполняется значительно быстрее, чем цикл с этой строковой командой, который мы организуем сами.

Точный смысл пары команд

```
REPE <строковая команда>
```

следующий (ZF – флаг нуля):

```
L: if CX=0 then goto L1;
   CX:=CX-1;
   <строковая команда>
   if ZF=1 then goto L;
L1:
```

Легко заметить, что префикс заставляет повторяться строковую команду такое число раз, которое указано в регистре CX, т. е. префикс использует этот регистр как счетчик цикла. Число повторений (оно всегда рассматривается как целое без знака) должно быть записано в регистр CX, конечно, до выполнения этой пары команд. Если это 0, то строковая команда не выполняется ни разу. Иначе префикс заставляет выполняться строковую команду и вычитает 1 из CX. И так до тех пор, пока в CX не окажется 0.

Однако это не единственная причина прекращения цикла. Как видно, после каждого выполнения строковой команды проверяется флаг нуля ZF. Если он равен 1, то цикл продолжается, но если после очередного выполнения строковой команды флаг ZF получит значение 0, то цикл тут же прекращается. А что означают значения 1 и 0 у флага ZF? Например, если строковая команда – это CMPS, то ZF=1 означает равенство сравниваемых элементов, а ZF=0 – неравенство. Таким образом, цикл продолжается, пока сравниваемые элементы равны, и прекращается, как только нашлась пара неравных элементов.

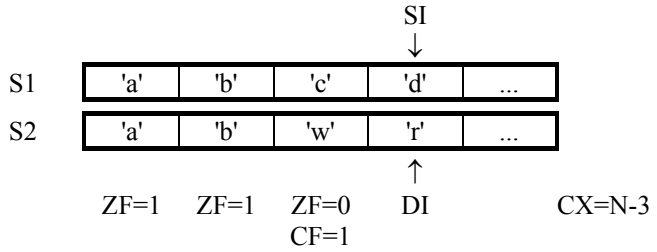
В целом, действие пары команд REPE CMPS можно описать так: повторяй сравнение элементов строк, пока они равны, но не более CX раз.

Итак, есть две причины выхода из цикла – либо строки просмотрены полностью и все их элементы оказались равными (тогда CX=0 и ZF=1), либо нашлась пара неравных элементов (тогда ZF=0, а CX может быть любым). По какой именно причине произошел выход, надо устанавливать после выхода из цикла анализом флага ZF с помощью команд условного перехода. Например, приведенные выше команды сравнения строк S1 и S2 (при просмотре вперед), можно переписать так:

```
    CLD         ;DF=0 (вперед)
    LEA SI,S1   ;DS:SI=начало S1
```

```
    LEA DI,S2   ;ES:DI=начало S2
    MOV CX,N    ;CX=длина строк
    REPE CMPSB  ;сравнивать, пока элементы равны (пока ZF=1)
      JE EQ     ;ZF=1 (S1=S2) ---> EQ
NOEQ: ...       ;ZF=0 (S1<>S2)
```

Рассмотрим, к примеру, сравнение следующих конкретных строк:



Первые две пары элементов равны, поэтому в ZF будет 1. Третьи элементы не равны, поэтому флаг ZF получает значение 0 и цикл прекращается. Поскольку после цикла ZF<>1, то мы переходим на метку NOEQ. Если бы все элементы строк были равны, то мы вышли бы по условию CX=0, а в ZF было бы значение 1, и в этом случае мы попали бы на метку EQ.

Теперь рассмотрим некоторые детали, которые важно знать при использовании префиксов повторения.

Во-первых, префикс повторения можно использовать не только при проверке строк на равно/не равно, но и при проверке на больше/меньше. В самом деле, если мы вышли из цикла при ZF=1, то значит, все элементы строк равны, но если оказалось ZF=0, то какая-то пара элементов строк различается. Но как различается? Напомним, что команда CMPS, как и команда CMP, меняет не только флаг ZF, но и другие флаги; например, в нашем случае будет CF=1, т. к. 'c'<'w' (коды символов – это беззнаковые целые). Поэтому проверкой других флагов и можно определить, какой из неравных элементов больше, а какой – меньше.

Если дописать в конец нашей группы команд сравнения строк команды

```
NOEQ: JA GREATER ;S1>S2 -> GREATER
LESS: ...        ;S1<S2
```

то получим проверку всех трех возможных исходов сравнения: при S1=S2 будет переход на метку EQ, при S1>S2 – на метку GREATER, а при S1<S2 – на метку LESS.

Во-вторых, по выходу из цикла в регистрах SI и DI оказываются адреса не тех элементов, на которых прекратилось сравнение, а следующих за ними. В нашем примере неравными оказались третьи элементы, но эти регистры будут показывать на четвертые элементы. Это объясняется тем, что команда CMPS после сравнения очередной пары элементов всегда меняет регистры SI и DI, равны эти элементы или нет.

В-третьих, после выхода из цикла в регистре CX всегда находится число элементов строк, оставшихся нерассмотренными, которые не сравнивались. Это очевидно, если вспомнить, что в начале в CX было число всех элементов строк и что после каждого сравнения из CX вычитается 1.

Нетрудно заметить, что после выхода из цикла регистры SI, DI и CX имеют такие значения, которые позволяют продолжить сравнение оставшихся частей строк (после пары неравных элементов). Этим, например, можно воспользоваться для подсчета числа неравных пар в строках:

```
      MOV AX,0  ;AX – число неравных пар
      CLD
      LEA SI,S1
      LEA DI,S2
      MOV CX,N
COMP: REPE CMPSB
      JE FIN    ;переход на FIN, если вышли из цикла
                ;при равной паре, т. е. дошли до конца строк
      INC AX    ;учесть очередную неравную пару
      CMP CX,0  ;продолжить сравнение "хвостов" строк,
      JNE COMP  ;если они не пусты
 FIN: ...
```

И, наконец, последнее замечание. Если до цикла значение регистра CX было нулевым, то значение флага ZF после цикла будет таким же, как и до цикла, поэтому по ZF нельзя определять причину выхода из "пустого" цикла. Это надо учитывать при сравнении пустых строк.

Теперь рассмотрим другой префикс повторения. Здесь все аналогично, но с заменой значения флага ZF на противоположное. А именно пара команд

```
REPNE <строковая команда>
```

выполняется следующим образом:

```
L: if CX=0 then goto L1;
   CX:=CX-1;
   <строковая команда>
   if ZF=0 then goto L;
L1:
```

Смысл префикса REPNE перед командой CMPS можно сформулировать так: повторяй сравнение элементов строк, пока они не равны, но не более CX раз. Значит, пока сравниваемые элементы строк различны, сравнение продолжается, но как только будет найдена пара равных элементов, цикл прекращается. Пара команд REPNE CMPS используется, когда надо найти первую (от начала или конца) пару равных элементов строк.

10.1.3. Другие строковые команды

Теперь рассмотрим другие строковые команды, но уже бегло, т. к. они во многом аналогичны команде сравнения строк.

Сканирование строки (SCAS, scan string): *SCASB*
SCASW

По команде SCASB содержимое регистра AL сравнивается с байтом памяти, абсолютный адрес которого указывает пара регистров ES:DI, после чего регистр DI автоматически устанавливается на соседний байт памяти:

```
AL=[ES:DI]? ; DI:=DI±1
```

(сравнение происходит так же, как в команде CMP AL,ES:[DI]; плюс берется при DF=0, а минус – при DF=1). Команда же SCASW сравнивает слова – из регистра AX и ячейки памяти, абсолютный адрес которого определяется парой ES:DI, после чего регистр DI также автоматически устанавливается на соседнее слово памяти:

```
AX=[ES:DI]? ; DI:=DI±2
```

Команда SCAS (в любом варианте) используется для поиска в строке элемента, равного заданному (в AL или AX) или отличного от заданного – в зависимости от того, какой префикс повторения поставлен перед ней:

REPNE SCASB – найти в строке первый элемент, равный AL
("повторяй сравнение, пока элементы не равны AL")

REPE SCASB – найти в строке первый элемент, отличный от AL
("повторяй сравнение, пока элементы равны AL")

Рассмотрим следующую задачу: в строке S из 500 символов, описанной в сегменте данных, надо заменить первое вхождение символа '*' на точку.

Для решения этой задачи запишем символ '*' в регистр AL и с помощью пары команд REPNE SCASB будем искать его в строке S. Если этот символ входит в строку, тогда, как и при использовании команды CPMS (см. выше), после выхода из цикла флаг ZF будет иметь значение 1, а регистр DI будет указывать на элемент строки, следующий за первым вхождением символа '*'. С учетом этого наша задача решается так:

```
     CLD                        ;просмотр строки вперед
     PUSH DS
     POP ES                     ;установить ES на сегмент данных
     LEA DI,S                   ;ES:DI=начало S
     MOV CX,500                 ;длина строки
     MOV AL,'*'                 ;символ для поиска
     REPNE SCASB                ;поиск первого вхождения '*' в S
     JNE FIN                    ;не входит --> FIN
     MOV BYTE PTR ES:[DI-1],'.' ;замена '*' на точку
FIN: ...
```

Пересылка строки (MOVS, move string): *MOVSB*
MOVSW

Команда MOVSB пересылает байт, а команда MOVSW – слово, абсолютный адрес которого задается парой регистров DS:SI, в ячейку, абсолютный адрес которой задается парой ES:DI, после чего значения регистров SI и DI

автоматически меняются так, чтобы они указывали на соседние элементы (правила изменения этих регистров такие же, как и в команде CMPS):

```
[DS:SI] => [ES:DI]; SI:=SI+d; DI:=DI+d
```

Флаги эти команды не меняют.

Команда MOVS (в любом варианте) пересылает только один элемент строки, но зато сразу настраивается на работу с соседним элементом. Поэтому осталось лишь зациклить эту команду, чтобы можно было переслать всю строку из одного места памяти в другое. Для такого зацикливания можно воспользоваться любым префиксом повторения, т. к. команда MOVS не меняет флаги и потому выход из цикла возможен только по одной причине – по CX=0, т. е. когда будут переписаны все элементы строки. Обычно перед командой MOVS указывается префикс с названием REP, т. е. просто "повторяй" без добавки "пока равно" или "пока не равно", т. к. они здесь малоосмысленны:

```
REP MOVSB ;повторяй пересылку (байтов) CX раз
```

Основное назначение команды MOVS – быстрая перепись содержимого одной области памяти в другую. Например, если в сегменте данных имеются массивы

```
X DW 100 DUP(?)
Y DW 100 DUP(?)
```

и требуется выполнить присваивание X:=Y, то сделать это можно так:

```
CLD         ;просмотр вперед
LEA SI,Y    ;DS:SI=начало Y ("откуда")
PUSH DS
POP ES      ;установка ES на сегмент данных
LEA DI,X    ;ES:DI=начало X ("куда")
MOV CX,100  ;сколько слов переписывать
REP MOVSW
```

Сохранение строки (STOS, store string): *STOSB*
STOSW

По команде STOSB в байт памяти, абсолютный адрес которого задается парой регистров ES:DI, записывается содержимое регистра AL, после чего значение регистра DI меняется на +1 при DF=0 или на -1 при DF=1. Команда STOSW записывает содержимое регистра AX в слово памяти, абсолютный адрес которого задается регистрами ES:DI, после чего меняет значение регистра DI на ±2. Флаги эти команды не меняют.

Перед командой STOS имеет смысл указывать только префикс REP, и тогда эту пару команд можно использовать для записи во все ячейки какой-то области памяти одной и той же величины – той, что указана в регистре AL или AX. Например, заполнить пробелами всю 40-символьную строку S из сегмента данных можно так:

```
MOV AL,' '    ;символ для записи
CLD           ;просмотр вперед
PUSH DS
POP ES
LEA DI,S      ;ES:DI=начало S
MOV CX,40     ;число заполняемых байтов
REP STOSB
```

Загрузка строки (LODS, load string): *LODSB*
LODSW

Команда LODSB (LODSW) записывает в регистр AL (AX) содержимое байта (слова) памяти, абсолютный адрес которого задается регистрами DS:SI, после чего меняет значение регистра SI на ±1 (на ±2). Флаги не меняются.

Указывать префикс повторения перед командой LODS бессмысленно. Обычно эта команда используется вместе с командой STOS для переписи строк, когда между считыванием и записью элементов строк над ними должна быть выполнена какая-то дополнительная операция. Например, если в сегменте данных имеются байтовые массивы X и Y из 101 знакового числа в каждом, тогда переписать все числа из X в Y с изменением их знака можно так:

```
   CLD           ;просмотр вперед
   LEA SI,X      ;DS:SI – "откуда" (для команды LODS)
   PUSH DS
   POP ES
   LEA DI,Y      ;ES:DI – "куда" (для команды STOS)
   MOV CX,101    ;сколько переписывать
L: LODSB         ;AL <- очередное число из X; SI:=SI+1
   NEG AL        ;изменение его знака
   STOSB         ;AL -> очередной элемент в Y; DI:=DI+1
   LOOP L
```

10.1.4. Команды загрузки адресных пар в регистры

Использование строковых команд и префиксов повторения позволяет существенно ускорить обработку строк, однако перед этими командами приходится выписывать достаточно много "установочных" команд (установить флаг направления, записать в регистр CX число повторений и т. д.). В определенной мере сократить число таких команд позволяют следующие две команды ПК, которые загружают в регистры адресные пары (указатели) и с помощью которых можно установить пары регистры DS:SI и ES:DI на обрабатываемые строки.

Загрузка указателя в DS (load pointer using DS): LDS r16,m32

В качестве второго операнда (m32) должен быть указан адрес двойного слова памяти, в котором находится пара seg:ofs, задающая абсолютный адрес некоторой ячейки памяти (из-за "перевернутого" представления двойных слов в памяти ПК часть ofs должна находиться в первом слове этого двойного слова, часть seg – во втором слове), а в качестве первого операнда – назва-

ние любого регистра общего назначения. Команда записывает в этот регистр смещение ofs, а в регистр DS – номер сегмента seg:

```
r16:=[m32], DS:=[m32+2]
```

Флаги команда не меняет.

Загрузка указателя в ES (load pointer using ES): LES r16,m32

Эта команда полностью аналогична команде LDS, только вместо регистра DS здесь используется регистр ES.

```
Пример:
DATA1 SEGMENT
   S1 DB 400 DUP(?)
  AS2 DD S2        ;AS2 = DATA2:S2
DATA1 ENDS
DATA2 SEGMENT
  S2 DB 400 DUP(?)
DATA2 ENDS
CODE SEGMENT
     ASSUME CS:CODE, DS:DATA1
     ...
;пусть в этот момент DS=DATA1
     CLD
     LEA SI,S1 ;DS:SI=начало S1
     LES DI,AS2    ;ES:DI=начало S2
     MOV CX,400
     REP MOVSB ;копирование строки S1 в строку S2
     ...
```

10.2. Строки переменной длины

В зависимости от того, меняются в процессе выполнения программы длины строк или нет, строки принято делить на строки переменной длины и строки фиксированной длины. Строки фиксированной длины – это обычные массивы, и как с ними работать, мы уже знаем. Сейчас нас будут интересовать строки переменной длины, причем только символьные строки (строки из байтов), поскольку обычно именно их имеют в виду, когда говорят про строки.

При работе со строкой переменной длины сразу возникает вопрос: сколько места отводить под нее? Ведь при составлении программы мы обязаны описать эту строку, явно указав, сколько места мы отводим под нее, но в это время длина строки еще неизвестна.

Возможный выход здесь – заранее узнать максимально возможную длину строки и отвести место в памяти под строку по этому максимуму. Например, если известно, что в любой момент строка S будет содержать не более 200 символов, тогда надо отвести под нее 200 байт:

```
S DB 200 DUP(?) ;длина(S)<=200
```

При этом символы, входящие в данный момент в строку, всегда будем располагать в начале выделенного участка памяти.

Отметим, что если заранее неизвестна и максимальная длина, тогда надо выкручиваться как-то по-другому, например, надо воспользоваться списками. Но о списках разговор дальше, а здесь мы будем рассматривать только тот случай, когда максимальная длина строки известна.

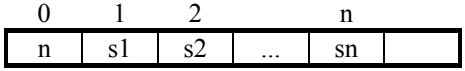
Поскольку длина строки может меняться, то надо как-то узнавать текущую длину строки. Решить эту проблему можно двояко.

Первый вариант – выбираем некоторый символ (скажем, с кодом 0), о котором заранее известно, что он не будет входить в строку, и договариваемся размещать его всегда за последним символом строки, т. е. используем его как признак конца строки:

При этом содержимое байтов за этим спецсимволом считается не относящимся к строке. Если длина строки меняется, то спецсимвол сдвигается.

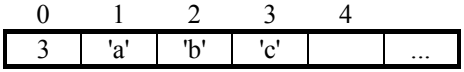
Этот способ представления строк переменной длины используется достаточно часто на практике (например, в языке Си). Но у него есть крупный недостаток: не просмотрев строку до конца, нельзя определить ее длину, а это может привести к лишней работе. Например, если надо проверить на равенство строку из 100 символов и строку из 50 символов, то ясно заранее, что эти строки не равны, однако при данном представлении строк мы можем узнать об этом, только просмотрев первые 50 символов.

Текущая длина строки легко узнается при другом представлении строк переменной длины, в котором начальная ячейка строки отводится для хранения текущей длины строки, а сами символы строки размещаются вслед за этой ячейкой:



Содержимое байтов, следующих за n-ым символом, считается не относящимся к текущему значению строки.

А сколько места отводить под длину строки (под n)? Это зависит от максимальной длины строки. Если эта длина больше 255, тогда, как минимум, надо отвести слово, а если она не превосходит 255, то достаточно и одного байта. Для определенности договоримся, что у нас строки будут содержать не более 255 символов, поэтому для хранения длины строк будем выделять один байт. При этом будем считать, что этот байт имеет индекс 0, тогда i-й символ строки окажется в байте с индексом i, т. е. адрес(S[i])=S+i. Например, строка 'abc' представляется таким образом:



Поскольку мы отводим место не только для символов строки, но для ее текущей длины, то этот дополнительный байт надо учитывать при описании

строки. Например, если заранее известно, что в строке не будет более 200 символов, то под нее надо отвести 201 байт:

```
S DB 201 DUP(?) ;длина(S)<=200
```

Отметим, что данное представление используется в языке Турбо Паскаль для строк типа string.

Рассмотрим пример обработки строк переменной длины при данном представлении их: требуется из строки S, длина которой не превосходит 200 и которая описана в сегменте данных, удалить первое вхождение символа '+', если такое есть.

Сначала, конечно, ищем '+' в S, для чего воспользуемся командой сканирования строки с подходящим префиксом повторения:

```
;поиск '+' в строке S
PUSH DS
POP ES
LEA DI,S+1    ;ES:DI = адрес S[1] (но не адрес S!)
CLD           ;просмотр строки вперед
MOV CL,S
MOV CH,0      ;CX=текущая длина строки (как слово)
MOV AL,'+'    ;искомый символ
REPNE SCASB   ;поиск '+' в S
JNE FIN       ;нет '+' в S -> на выход
```

Остановимся на минутку и отметим следующее.

Если '+' входит в S, тогда в этот момент пара регистров ES:DI будет указывать на символ, следующий за '+', а в регистре CX будет находиться число символов строки, следующих за '+'.

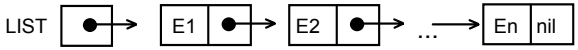
Теперь надо удалить '+'. Что это значит? Поскольку оставлять пустое место нельзя, то надо сдвинуть "хвост" строки на одну позицию влево. Для этого можно воспользоваться командой пересылки строки MOVS. Перенести нужно CX элементов, направление просмотра – вперед, поэтому менять регистр CX и флаг DF не надо. В команде MOVS перенос происходит от адреса DS:SI к адресу ES:DI, поэтому устанавливаем регистр SI на символ, следующий за '+' ("откуда"), а регистр DI – на сам символ '+' ("куда"). Кроме того, надо уменьшить длину строки на 1. В итоге получаем такое окончание решения нашей задачи:

```
;удаление '+' из S
     MOV SI,DI    ;DS:SI - откуда
     DEC DI       ;ES:DI - куда
     REP MOVSB    ;сдвиг "хвоста" строки на 1 позицию влево
     DEC S        ;уменьшение длины строки на 1
FIN: ...
```

10.3. Списки

В этом разделе рассматриваются способы представления и методы работы со списками в машинных программах.

Списком (линейным, однонаправленным) называется последовательность звеньев, которые могут размещаться в произвольных местах памяти и в каждом из которых указывается элемент списка (Ei) и ссылка на следующее звено (изображена стрелкой):



В последнем звене размещается специальная "пустая" ссылка nil, указывающая на конец списка. Ссылка на первое звено списка хранится в некоторой переменной LIST; если список пуст, то ее значением является nil.

Возможность размещать звенья списков в любых местах памяти обуславливают плюсы и минусы списков (по сравнению с другим способом представления последовательностей данных – массивами, где соседние элементы располагаются в соседних ячейках памяти). Достоинством списков является то, что их длина заранее не фиксируется, что под них надо отводить ровно столько места, сколько требуется в текущий момент, и что удаление и вставка элементов реализуются просто и быстро – заменой ссылок в одном-двух звеньях. К недостаткам же списков относится лишний расход памяти для хранения ссылок и то, что добраться до какого-то элемента списка можно лишь просмотрев все предыдущие элементы.

В большинстве ЭВМ, в частности в ПК, не предусмотрено никакого стандартного способа представления списков и нет специальных команд для работы со списками. Все это приходится реализовывать самим авторам программ. Мы рассмотрим один из возможных вариантов того, как это можно сделать.

10.3.1. Представление списков

Начнем с представления списков в памяти ЭВМ.

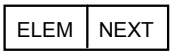
Прежде всего рассмотрим, где размещаются звенья списков. Для их хранения обычно отводят специальную область памяти, называемую областью списков (list space) или кучей (heap); мы будем использовать термин "куча". Размер кучи зависит от того, сколь много списков будет использоваться в программе. Договоримся для определенности, что для наших программ достаточно кучи размером до 64 Кб, поэтому на кучу отведем один сегмент памяти (позже будет показано, как можно описать этот сегмент).

Договоримся также, что на начало кучи постоянно указывает сегментный регистр ES. Если внутри кучи звено списка имеет адрес (смещение) A, то абсолютный адрес этого звена задается адресной парой ES:A. Именно эту пару и надо указывать в командах, когда хотим получить доступ к звену.

Теперь о представлении ссылок. Ссылка на звено – это, конечно, адрес данного звена. Но какой адрес здесь имеется в виду – смещение A или адресная пара H:A, где H – начало кучи? Поскольку во всех этих адресных парах первый элемент (H) один и тот же, то его можно явно не указывать, а лишь подразумевать. Поэтому ради экономии памяти договоримся считать ссыл-

кой на звено лишь смещение A, т. е. адрес этого звена, отсчитанный от начала кучи. В связи с этим ссылками у нас будут 16-разрядные адреса.

Под каждое звено отводим несколько соседних байтов памяти, в первых из которых размещаем соответствующий элемент списка ELEM, а в остальных – ссылку на следующее звено NEXT:



Размер звена зависит от размера элементов списка. Для определенности будем считать, что элементами списка являются данные размером в слово, поэтому на элемент отводим 2 байта. А поскольку ссылка также занимает два байта, то всего на звено отводим 4 байта, т. е. двойное слово.

При программировании на ЯА звенья списка удобно рассматривать как структуры следующего типа:

```
NODE STRUC  ;тип звена списка
ELEM DW ?   ;элемент списка
NEXT DW ?   ;ссылка на следующее звено
NODE ENDS
```

Поэтому, если нам надо получить доступ к полям звена, который имеет адрес A, то в командах мы будем указывать такие выражения:

```
ES:A.ELEM – поле с элементом
ES:A.NEXT – поле с адресом следующего звена
```

Теперь о пустой ссылке nil. Обычно в этом качестве используется адрес 0. Так будем делать и мы. При этом удобно описать в программе этот адрес как константу:

```
NIL EQU 0
```

и далее пользоваться именем NIL, а не нулем. Так более наглядно.

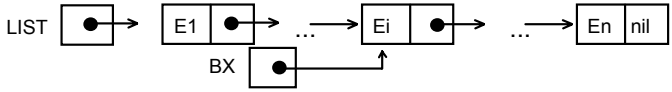
И, наконец, последнее. Ссылки на первые звенья списков будем, как правило, хранить в переменных из сегмента данных. Эти переменные имеют размер слова и описываются как обычно:

```
LIST DW ?
```

Зная адрес первого звена списка, можно по ссылкам добраться до любого звена списка, поэтому договоримся о следующем: указать список – значит указать имя переменной (например, LIST), в которой находится адрес первого звена списка.

10.3.2. Операции над списками

При работе со списком приходится последовательно просматривать его звенья. Поэтому нам надо знать адрес текущего звена – того, которое мы обрабатываем в данный момент. Для хранения этого адреса будем использовать какой-нибудь регистр-модификатор, например, BX. Если текущим является i-е звено списка, тогда имеем следующую картинку:



Если надо сделать текущим какое-то иное звено, то мы обязаны в регистр BX занести адрес этого звена.

В регистре BX хранится только смещение текущего звена – адрес, отсчитанный от начала кучи. Поэтому, если мы хотим добраться до самого звена, то надо использовать выражение ES:[BX]; писать же просто [BX] нельзя, ибо, согласно правилам ПК, такой адрес по умолчанию будет сегментироваться по регистру DS. Что же касается доступа к полям текущего звена, то для этого используются выражения ES:[BX].ELEM и ES:[BX].NEXT.

Теперь рассмотрим, как реализуются операции над списками. Начнем с простейших операций: анализ текущего элемента, переход к следующему звену и проверка на конец списка.

Анализ текущего элемента списка

Пусть надо определить, равен ли элемент из текущего звена списка значению некоторой переменной X, и, если да, перейти на метку EQ. Реализуется это так (в комментариях указана запись этих действий на языке Паскаль):

```
MOV AX,ES:[BX].ELEM
CMP AX,X   ;if bx↑.elem=x then goto eq
JE EQ
```

Переход к следующему звену списка

Пусть регистр BX указывает на некоторое звено списка и требуется перейти к следующему звену списка. Согласно нашей договоренности, это означает, что в регистр BX надо заслать адрес следующего звена. Адрес же этот находится в поле NEXT текущего звена, поэтому его надо перенести в BX:

```
MOV BX,ES:[BX].NEXT ;bx:=bx↑.next
```

Проверка на конец списка

Может оказаться так, что следующего звена нет, тогда в регистр BX попадет пустая ссылка nil. Поэтому сравнением BX с nil и можно определить, дошли мы до конца списка или нет:

```
CMP BX,NIL
JE LIST_END  ;if bx=nil then goto list_end
```

Так реализуются простейшие, элементарные операции над списками. На их основе можно уже построить более крупные операции – просмотр списка, удаление элемента из списка, вставка нового элемента в список. Рассмотрим, как реализуются эти операции, на конкретных примерах.

Пример 1 (поиск элемента в списке)

Пусть имеются список LIST и переменная X. Требуется определить, входит ли в этот список элемент, равный X, и записать в регистр AL ответ 1 (входит) или 0.

Идея решения ясна: последовательно просматриваем звенья списка и сравниваем находящиеся в них элементы с X.

```
     MOV AL,0
     MOV CX,X          ;искомая величина
     MOV BX,LIST       ;bx – nil или адрес 1-го звена
  L: CMP BX,NIL
     JE NO             ;конец списка -> no
     CMP ES:[BX].ELEM,CX
     JE YES            ;bx↑.elem=x -> yes
     MOV BX,ES:[BX].NEXT   ;bx – адрес следующего звена
     JMP L             ;в цикл
YES: MOV AL,1
 NO: ...
```

Пример 2 (вставка элемента в список)

Требуется вставить в начало списка LIST новый элемент X.

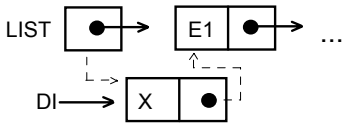
Здесь прежде всего надо отвести место под новое звено. В языке Паскаль для этого используется стандартная процедура New. К сожалению, в ЯА такой процедуры нет. Но ее можно определить самим. Как это сделать, мы рассмотрим чуть позже, а пока будем считать, что такая процедура уже имеется и мы можем ею пользоваться. Уточним только ее действие.

Обращение к ней выглядит так:

```
CALL NEW   ;new(di)
```

У процедуры нет входных параметров. Она отыскивает в куче свободное место (двойное слово) и его адрес возвращает через регистр DI. Это место можно занять новым звеном.

После этого надо заполнить новое звено, т. е. в поле ELEM записать величину X, а в поле NEXT – ссылку на бывшее первое звено (она берется из LIST), и далее в LIST надо записать адрес нового звена (берется из DI):



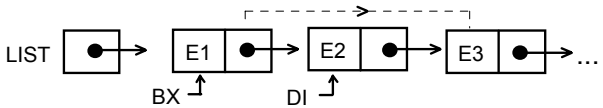
С учетом всего сказанного наша задача решается следующим образом:

```
CALL NEW      ;new(di)
MOV AX,X
MOV ES:[DI].ELEM,AX   ;di↑.elem:=x
MOV AX,LIST
MOV ES:[DI].NEXT,AX   ;di↑.next:=list
MOV LIST,DI   ;list:=di
```

Пример 3 (удаление элемента из списка).

Требуется удалить из списка LIST второй элемент, если такой есть.

Пусть в списке есть второй элемент. Тогда сделаем так, чтобы регистр BX указывал на первое звено, а регистр DI – на второе звено:



Для исключения второго звена из списка надо изменить ссылку в первом звене – записать сюда ссылку на третье звено. Сделать это просто: ссылка на третье звено находится во втором звене, поэтому ее отсюда надо перенести в первое звено.

Однако лишь поменять ссылку в первом звене мало. Второе звено занимает место в памяти, а теперь оно стало ненужным, поэтому следует освободить это место. В языке Паскаль в этих целях используется стандартная процедура Dispose. В ЯА ее нет, однако ее можно определить самим. Чуть позже мы определим такую процедуру, а пока будем считать, что она уже имеется и мы можем ею пользоваться. Уточним лишь правила обращения к ней:

```
DI:=адрес ненужного звена
CALL DISPOSE ;dispose(di)
```

Сначала надо записать в регистр DI адрес звена, ставшего ненужным, а затем обратиться к процедуре, которая каким-то образом учтет, что это место кучи стало свободным.

С учетом всего сказанного наша задача решается так:

```
    CMP LIST,NIL
    JE FIN                  ;list=nil -> fin
    MOV BX,LIST             ;bx – адрес 1-го звена
    MOV DI,ES:[BX].NEXT     ;di – nil или адрес 2-го звена
    CMP DI,NIL
    JE FIN                  ;dx=nil -> fin
    MOV AX,ES:[DI].NEXT
    MOV ES:[BX].NEXT,AX     ;bx↑.next:=di↑.next
    CALL DISPOSE            ;dispose(di)
FIN: ...
```

Вот так реализуются основные операции над списками. Используя их как образец, можно реализовать любые другие операции над списками.

10.3.3. Организация кучи

Теперь займемся организацией кучи и описанием процедур New и Dispose. Напомним, что при программировании на ЯА все это мы обязаны делать сами, никто за нас этого делать не будет. Мы рассмотрим один из возможных вариантов того, как это можно сделать (этот вариант используется в трансляторах языка Лисп).

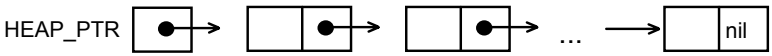
Список свободной памяти

Как уже было сказано, для размещения звеньев всех списков, используемых в программе, необходимо отвести специальную область памяти, называемую кучей. При выполнении программы одни ячейки кучи оказываются занятыми звеньями списков, а другие – пока свободны. Поскольку программа

может то "захватывать" ячейки кучи, то "отказываться" от них, в общем случае занятые и свободные ячейки кучи идут вперемежку. Возникает вопрос: как в таких условиях определять, какие ячейки свободны?

Для этого следует все свободные ячейки кучи объединить в один список, который принято называть списком свободной памяти (ССП). Начало этого списка указывается в некоторой фиксированной ячейке, скажем, с именем HEAP_PTR (heap pointer, указатель кучи). Значит, помимо списков, с которыми работает программа, заводится еще один список. Используется же ССП так: когда программе нужно место под новое звено, то оно выделяется из этого ССП, а когда программа отказывается от какого-то звена, то это звено добавляется к ССП.

Поскольку у нас размеры звеньев списков фиксированы (под каждое из них всегда отводится двойное слово), то имеет смысл объединять в ССП именно двойные слова. Если при этом ссылку на следующее звено ССП хранить во втором слове звена, то мы получим обычный список:



(содержимое левых полей произвольно), и потому способы работы с этим списком такие же, как и с обычными списками.

Где размещать переменную HEAP_PTR? Вообще говоря, где угодно. Но лучше всего отвести ей место в самом начале кучи – в ячейке с относительным адресом 0. Почему? А потому, что тогда ни одно звено не будет размещаться в этой ячейке и не будет иметь адрес 0. Значит, этот адрес остается свободным и его можно использовать для представления пустой ссылки nil, что мы и делали. Если же разместить HEAP_PTR в другом месте, то все равно эту ячейку кучи нельзя было бы занимать, и она пропала бы зря.

Описание сегмента кучи

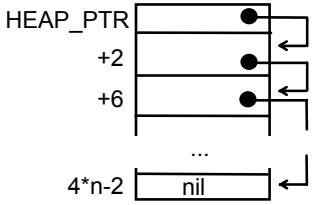
Поскольку куча – это один из сегментов памяти, используемый программой, то его надо описать в программе. Пусть мы решили, что куча в целом должна вмещать n звеньев (двойных слов). Тогда удобно ввести константу для этого числа и использовать ее при описании сегмента кучи:

```
HEAP_SIZE EQU n         ;n – число звеньев в куче: 4*n<64K
     HEAP SEGMENT       ;сегмент кучи
 HEAP_PTR DW ?          ;ячейка с начальным адресом ССП
          DD HEAP_SIZE DUP(?)   ;собственно куча
                                ;(n двойных слов)
     HEAP ENDS
```

Инициализация кучи

Мы договорились, что сегментный регистр ES указывает на начало сегмента кучи. Но, естественно, загрузку этого регистра обязаны сделать мы сами в начале работы программы.

Кроме того, вначале все ячейки кучи свободны, однако они не объединены в ССП, и работать с кучей пока еще нельзя. Поэтому в начале программы надо связать все ячейки (двойные слова) кучи в единый список и записать в HEAP_PTR ссылку на начало этого списка. При этом совершенно безразлично, в каком именно порядке будут объединены эти ячейки. Например, их можно связать сверху вниз, чтобы верхняя ячейка кучи была первым звеном этого ССП, а нижняя – последним:



Оба указанных действия – загрузку регистра ES и построение исходного ССП – мы опишем в виде одной процедуры:

```
INIT_HEAP PROC FAR
          PUSH SI
          PUSH BX
          PUSH CX
;установка ES на начало сегмента кучи
          MOV CX,HEAP
          MOV ES,CX
;объединение всех двойных слов кучи в ССП
          MOV CX,HEAP_SIZE      ;число звеньев в ССП
          MOV BX,NIL         ;ссылка на построенную часть ССП
          MOV SI,4*HEAP_SIZE-2  ;адрес нового звена ССП
   INIT1: MOV ES:[SI].NEXT,BX   ;добавить в начало
                                ;построенного
          MOV BX,SI             ; ССП новое звено
          SUB SI,4              ;SI – на двойное слово "выше"
          LOOP INIT1
          MOV ES:HEAP_PTR,BX ;HEAP_PTR – на начало ССП
          POP CX
          POP BX
          POP SI
          RET
INIT_HEAP ENDP
```

Обратиться к этой процедуре надо в начале программы, после чего уже можно использовать процедуры NEW и DISPOSE.

Процедура Dispose

К этой процедуре программа обращается, когда она отказывается от некоторого звена. Адрес этого звена, как мы договорились, передается через регистр DI. Это звено становится свободным, поэтому его надо просто присое-

динить к ССП. Проще всего звено добавить в начало ССП, а как вставлять звено в начало списка, мы уже знаем.

```
;на входе: DI – адрес ненужного звена
DISPOSE PROC FAR
        PUSH ES:HEAP_PTR
        POP ES:[DI].NEXT    ;di↑.next:=heap_ptr
        MOV ES:HEAP_PTR,DI  ;heap_ptr:=di
        RET
DISPOSE ENDP
```

Процедура New

К этой процедуре программа обращается, когда нужно место для нового звена какого-то списка. Это место берется из ССП: от этого списка отщепляется одно из звеньев и оно отдается программе. Проще всего отдать первое звено ССП, что и будем делать. Адрес этого звена, согласно договоренности, процедура должна вернуть через регистр DI. Поэтому процедура должна в регистр DI перенести значение HEAP_PTR, а затем в эту переменную записать ссылку на второе звено ССП (она берется из первого звена).

Однако здесь надо учитывать особый случай: ССП может быть пустым. Это может случиться, если программа захватила все ячейки кучи и обратно их не отдает. Что делать в этом случае? Надо просто прекратить выполнение программы, т. к. она все равно не сможет правильно продолжить свою работу (увеличить по ходу дела размер кучи, как правило, сложно). Так мы и сделаем: при пустом ССП сообщим об ошибке и завершим выполнение программы.

С учетом всего сказанного получаем такое описание процедуры New:

```
;на выходе: DI – адрес свободного звена
 NEW PROC FAR
     MOV DI,ES:HEAP_PTR  ;di – nil или адрес 1-го звена ССП
     CMP DI,NIL
     JE EMPTY_HEAP       ;пустой ССП -> EMPTY_HEAP
     PUSH ES:[DI].NEXT
     POP ES:HEAP_PTR     ;heap_ptr – на 2-е звено ССП
     RET
EMPTY_HEAP:              ;реакция на пустую кучу
     LDS DX,CS:AERR      ;DS:DX – адрес сообщения об ошибке
     OUTSTR              ;вывод этого сообщения
     FINISH              ;останов программы
AERR DD ERR              ;абс. адрес сообщения
 ERR DB 'Ошибка в NEW: исчерпание кучи','$'
 NEW ENDP
```

В заключение рассказа об организации кучи сделаем пару замечаний.

Приведенные варианты процедур NEW и DISPOSE рассчитаны только на случай, когда звенья всех списков имеют один и тот же размер. Если же в списках используются звенья разных размеров, то нужны более изощренные способы работы со ССП. Но мы их не будем рассматривать, это не тема данной книги.

Поскольку ни в ПК, ни в ЯА не предусмотрено никаких средств для работы со списками и кучей, то в любой программе, где используются списки, приходится явно описывать рассмотренные в этом разделе процедуры, сегмент кучи, тип NODE, константу NIL. И если таких программ много, то, чтобы не повторяться, имеет смысл все эти описания собрать в одном файле, скажем с названием LISTS.ASM, а затем подключать их к программам с помощью директивы INCLUDE LISTS.ASM.

11. МАКРОСРЕДСТВА

В данной главе рассматриваются макросы, блоки повторения и средства условного ассемблирования, расширяющие возможности языка ассемблера.

11.1. Макроязык

Нередко бывает полезным предварительное (до начала трансляции) преобразование текста программы. Например, может потребоваться, чтобы какой-то фрагмент программы был продублирован несколько раз или чтобы в зависимости от некоторых условий в тексте программы были сохранены одни фрагменты и удалены другие. Подобную возможность предоставляют так называемые макросредства. Расширение языка ассемблера за счет этих средств обычно называют макроязыком.

Программа, написанная на макроязыке, транслируется в два этапа. Сначала она переводится на, так сказать, чистый язык ассемблера, т. е. преобразуется к виду, где нет никаких макросредств. Этот этап называется этапом макрогенерации, его осуществляет специальный транслятор – макрогенератор. На втором этапе полученная программа переводится на машинный язык. Это этап ассемблирования, его осуществляет ассемблер.

Как видно, трансляция программы, написанной на макроязыке, занимает больше времени, чем трансляция программы на "чистом" языке ассемблера. Но зато макроязык предоставляет больше возможностей, чем просто язык ассемблера, и потому писать программы, особенно большие, на макроязыке проще.

Макрогенератор и ассемблер могут взаимодействовать двояко. Во-первых, они могут действовать независимо друг от друга: сначала текст программы обрабатывает макрогенератор и только затем начинает работать ассемблер. Такой способ взаимодействия достаточно прост для понимания и для реализации и используется довольно часто, однако у него есть серьезный недостаток: макрогенератор, работая до ассемблера, не может воспользоваться информацией, извлекаемой из текста программы ассемблером. Например, в макросредствах нельзя использовать то, что после директивы

```
N EQU 10
```

имя N обозначает число 10, т. к. эта директива относится к "чистому" языку ассемблера и потому будет обработана только ассемблером. Из-за подобного рода ограничений приходится усложнять макроязык.

Во-вторых, макрогенератор и ассемблер могут действовать совместно, чередуя свою работу: первым каждое предложение программы просматривает макрогенератор; если это конструкция собственно макроязыка, то макрогенератор соответствующим образом преобразует ее в группу предложений "чистого" языка ассемблера и тут же передает их на обработку ассемблеру, а если это обычное предложение "чистого" языка ассемблера, то макрогенера-

тор сразу передает его ассемблеру. В таких условиях в конструкциях макроязыка уже можно ссылаться на объекты (например, константы) "чистого" языка ассемблера.

При "чередующемся" способе взаимодействия макрогенератора и ассемблера они фактически являются не двумя независимыми трансляторами, а частями одного транслятора, который принято называть макроассемблером.

Именно такой макроассемблер используется в рассматриваемой нами системе MASM. Этим и объясняется полное название ее входного языка – язык макроассемблера (macroassembler language, MASM). Однако в дальнейшем мы по-прежнему будем называть его языком ассемблера (сокращенно – ЯА), используя термин "макроязык" для обозначения только набора макросредств, а термин "макрогенератор" для обозначения той части макроассемблера, что занимается обработкой этих макросредств. Кроме того, термином "окончательная программа" мы будем называть ту программу, которая получается после работы макрогенератора, которая затем переводится (собственно) ассемблером на машинный язык и выполняется.

11.2. Блоки повторения

Иногда в некотором месте программы приходится выписывать несколько раз подряд один и тот же (или почти один и тот же) фрагмент, и хотелось бы, чтобы мы сами выписывали этот фрагмент только раз, а макрогенератор размножал его нужное число раз. Такая возможность предусмотрена в ЯА, и реализуется она с помощью блоков повторения (repeat blocks).

Блок повторения имеет следующую структуру:

```
<заголовок>
<тело>
ENDM
```

Здесь <тело> – любое число любых предложений (в частности, ими могут быть снова блоки повторения), а ENDM – директива, указывающая на конец тела и всего блока повторений. Встречая в исходном тексте программы такой блок, макрогенератор подставляет вместо него в окончательную программу несколько копий тела.

При дублировании тело может выписываться без каких-либо изменений, а может копироваться и с модификациями. Как именно происходит дублирование, сколько копий создается – все это зависит от заголовка блока. Имеется три разновидности заголовка, в связи с чем различают три варианта блока повторения: REPT-блоки, IRP-блоки и IRPC-блоки.

11.2.1.REPT-блоки

Этот тип блоков повторения записывается следующим образом:

```
REPT k
<тело>
ENDM
```

Здесь k – константное выражение с неотрицательным значением. Это выражение должно быть таким, чтобы можно было вычислить его сразу (например, в нем не должно быть ссылок вперед). Вычислив значение k, макрогенератор создает k точных копий тела блока и подставляет их в окончательный текст программы. Например, по блоку

```
REPT 3
SHR AX,1
ENDM
```

будет построен следующий фрагмент окончательной программы:

```
SHR AX,1
SHR AX,1
SHR AX,1
```

Другой пример (слева указан фрагмент исходной программы, справа – построенный по нему фрагмент окончательной программы):

```
N EQU 6              N EQU 6
  REPT N-4             DB 0,1
  DB 0,1      ⇒        DW ?
  DW ?                 DB 0,1
  ENDM                 DW ?
```

Отметим, что в блоках повторения довольно часто используется директива присваивания (=). Например, описать 100-байтовый массив X, элементы которого имеют начальные значения от 0 до 99, можно так (справа указан текст окончательной программы, который фактически эквивалентен директиве
X DB 0,1,2,3,...,99):

```
X DB 0                 X DB 0
K=0                    K=0
  REPT 99              K=K+1
  K=K+1                  DB K
  DB K         ⇒       K=K+1      } 99 таких пар
  ENDM                   DB K
                         ...
```

11.2.2.IRP-блоки

Блоки повторения этого типа имеют следующий вид:

```
IRP p,<v1, ..., vk>
<тело>
ENDM
```

Замечание. Уголки в записи <v1, ... ,vk> – это явно указываемые символы, а не метасимволы.

Здесь p – некоторое имя, оно играет роль формального (фиктивного) параметра и может использоваться в предложениях тела; vi – это фактические параметры; это любые тексты (возможно, и пустые), но, чтобы не было путаницы, они должны быть сбалансированы по кавычкам и не должны содержать запя-

тые, точки с запятой и уголки вне кавычек (если это ограничение надо нарушить, то следует воспользоваться макрооператорами – см. разд. 11.2.4). Параметры vi перечисляются через запятую, а вся их совокупность обязательно заключается в угловые скобки.

Встречая такой блок, макрогенератор заменяет его на k копий тела (по одной на каждый фактический параметр), причем в i-й копии все вхождения имени p заменяются на vi. Например:

```
IRP REG,<AX,CX,SI>          PUSH AX
PUSH REG            ⇒       PUSH CX
ENDM                        PUSH SI
```

Отметим, что формальный параметр локализуется в теле блока (им нельзя пользоваться вне блока) и может быть любым именем. Если оно совпадает с именем другого объекта программы, то в теле блока оно обозначает именно параметр, а не этот объект. Например, в блоке

```
IRP BX,<1,5>
ADD AX,BX
ENDM
```

имя BX обозначает параметр, а не регистр, поэтому по данному блоку будет построен следующий фрагмент окончательной программы:

```
ADD AX,1
ADD AX,5
```

Замены формального параметра на фактические – это чисто текстуальные подстановки, без учета смысла: просто один участок текста (p) заменяется на другой (vi). При этом параметром p можно обозначить любую часть предложения (в частности, участки комментария) или даже целиком все предложение (однако два и более предложений он не может обозначать), лишь бы после замены p на vi получались правильные предложения ЯА. Например:

```
IRP Q,<DEC WORD PTR,L: INC>          DEC WORD PTR W
Q W                                  JMP M2
JMP M2                         ⇒   L: INC W
ENDM                                 JMP M2
```

Отметим также, что в теле блока повторения заменяется только формальный параметр, другие имена (например, имена констант) переносятся в копии тела без изменений. Например:

```
N EQU 1              N EQU 1
  IRP P,<A,B>  ⇒     A EQU N       (но не A EQU 1 )
P EQU N              B EQU N
  ENDM
```

Другие особенности записи формальных и фактических параметров будут рассмотрены в разд. 11.2.4.

11.2.3. IRPC-блоки

Блоки этого типа записываются следующим образом:

```
IRPC p,s1...sk
<тело>
ENDM
```

И здесь p – формальный параметр, а вот si – это символы. Это могут быть любые символы, кроме пробелов и точек с запятой (считается, что с точки с запятой начинается комментарий, а пробел заканчивает операнд; если надо указать точку с запятой или пробел, то всю последовательность символов следует заключить в угловые скобки – см. разд. 11.2.4). Встречая IRPC-блок, макрогенератор заменяет его на k копий тела блока (по одной на каждый символ), причем в i-й копии все вхождения параметра p будут заменены на символ si.

Пример:

```
IRPC D,17W          ADD AX,1
ADD AX,D     ⇒      ADD AX,7
ENDM                ADD AX,W
```

11.2.4. Макрооператоры

При использовании блоков повторения (и макросов, которые будут рассмотрены чуть позже) возникает ряд проблем с записью их формальных и фактических параметров. Эти проблемы решаются с помощью так называемых макрооператоров – операторов, разрешенных к применению только в конструкциях макроязыка.

Макрооператор &

Рассмотрим следующий блок повторения и построенные по нему копии:

```
  IRP W,<VAR1,VAR6>        VAR1 DW ?
W DW ?                ⇒    VAR6 DW ?
  ENDM
```

Здесь параметр W обозначает имя переменной целиком. Но фактические имена (VAR1 и VAR6) различаются лишь последним символом, поэтому было бы разумнее объявить параметром только этот символ, а не все имя. Но если так и сделать:

```
IRP W,<1,6>
VARW DW ?
ENDM
```

то получится неоднозначность: становится непонятным, когда W обозначает формальный параметр, а когда саму букву W (почему в VARW надо W заменять на 1 и 6, а в DW не надо?). Во всех предыдущих примерах мы не сталкивались с такой проблемой, т. к. формальные параметры легко выделялись из окружающего текста благодаря ограничителям (пробелам, запятым и т. п.), стоящим слева и справа от них. Но если рядом с параметром стоит имя

или число, то границы параметра становятся неопределяемыми. В подобной ситуации следует между параметром и соседним с ним числом или именем поставить символ & (A&W, 1&W&B и т. п.). Например, наш блок повторения должен быть записан следующим образом:

```
      IRP W,<1,6>
VAR&W DW ?
      ENDM
```

Назначение знака & – указать границу формального параметра, выделить его из окружающего текста, при этом в окончательный текст программы он не попадает. (Если & поставить не около параметра, то он будет просто опущен.)

Макрооператор & используется не только тогда, когда формальный параметр "сливается" с соседними именами и числами, но и когда его надо указать внутри строк. Дело в том, что макрогенератор игнорирует вхождения формального параметра в строки, и чтобы обратить его внимание на эти вхождения, перед параметром в строках надо ставить знак & (а если не ясна его правая граница, то & надо указывать и после параметра). Например:

```
IRPC A,"<             DB 'A,","B'
DB 'A,&A,&A&B'   ⇒    DB 'A,<,<B'
ENDM
```

И еще одна особенность макрооператора &: если рядом поставить несколько знаков &, то макрогенератор удалит только один из них. Это сделано специально, учитывая возможность вложенности блоков повторений (и/или макросов). Например:

```
IRPC P1,AB      IRPC P2,HL          INC AH
IRPC P2,HL      INC A&P2            INC AL
INC P1&&P2  ⇒   ENDM          ⇒     INC BH
ENDM            IRPC P2,HL          INC BL
ENDM            INC B&P2
                ENDM
```

Встретив в тексте исходной программы блок повторения, указанный в левой колонке, макрогенератор сначала создаст первую копию тела внешнего блока, в котором все вхождения его формального параметра P1 будут заменены на символ A (см. три верхние строчки средней колонки). При этом из двух подряд стоящих в команде INC знаков & будет удален только один, и оставшийся знак & будет отделять формальный параметр P2 внутреннего блока от стоящей слева буквы A (если бы был только один знак &, то эта команда имела бы вид INC AP2, и потому запись AP2 не воспринималась бы как состоящая из двух частей – A и P2). Поскольку в полученной копии остались конструкции макроязыка, то макрогенератор продолжает свою работу (это общее правило: макрогенератор работает, пока не получится текст на "чистом" языке ассемблера) и "раскручивает" внутренний блок, получая уже

окончательный текст (см. две верхние строчки в правой колонке). Далее создается вторая копия внешнего блока, которая обрабатывается аналогично.

Макрооператор <>

Как было сказано, фактические параметры IRP-блока не должны содержать запятые, точки с запятой и уголки, а во втором операнде IRPC-блока нельзя указывать пробелы и точки с запятой. Эти ограничения связаны с тем, что иначе возможна путаница: например, если внутри фактического параметра IRP-блока указать запятую (скажем: 1,2), тогда будет непонятным, что означает эта запись – то ли два параметра, разделенных запятой, то ли один параметр. Так вот, если надо нарушить указанные ограничения, тогда весь фактический параметр IRP-блока или всю последовательность символов в IRPC-блоке надо заключить в угловые скобки (например: <1,2>), причем текст внутри этих скобок должен быть сбалансирован по уголкам. При этом считается, что внешние угловые скобки не относятся к параметру или последовательности, что они лишь указывают их границы.

Примеры:

```
IRP VAL,<<1,2>,3>          DB 1,2
DB VAL              ⇒      DB 3
ENDM
IRPC S,<A;B>               DB 'A'
DB '&S'             ⇒      DB ';'
ENDM                       DB 'B'
```

Макрооператор !

А что делать, если внутри фактического параметра надо указать непарный уголок или кавычку? Для задания этих и других спецсимволов (вне или внутри угловых скобок) предусмотрен следующий макрооператор:

```
!<символ>
```

Смысл этой записи: сам символ ! "погибает" (не переносится в окончательный текст), но следующий за ним символ трактуется как обычный символ, а не как символ, играющий какую-то специальную роль. Например:

```
IRP X,<A!>B,Привет!, ПК!!>        DB 'A>B'
DB '&X'                     ⇒     DB 'Привет, ПК!'
ENDM
```

Макрооператор ! можно использовать только при записи фактических параметров IRP-блоков (и макросов), тогда как в последовательности символов (во втором операнде) IRPC-блока знак ! рассматривается как обычный символ. Указывать в этой последовательности уголки можно сколько угодно раз, если эта последовательность не начинается с открывающей угловой скобки, а если начинается – пока нет баланса угловых скобок.

Макрооператор %

В макроязыке есть еще один макрооператор, используемый при записи фактических параметров IRP-блоков (и макросов):

```
%<константное выражение>
```

Встретив такую конструкцию в фактическом параметре, макрогенератор вычисляет указанное выражение и подставляет его значение вместо всей этой конструкции. Например:

```
K EQU 4
  ...
  IRP A,<K+1,%K+1,W%K+1>        DW K+1
  DW A                    ⇒     DW 5
  ENDM                          DW W5
```

Отметим, что вложенность макрооператоров % не допускается (например, в конструкции %5-%K будет зафиксирована ошибка "неописанное имя %K") и что концом константного выражения считается первый символ (например, запятая, угловая скобка или знак равенства), который не может по синтаксису входить в константные выражения (например, при значении 4 у константы K параметр %K-1+K=K будет преобразован в 7=K).

Отметим также, что в последовательности символов (во втором операнде) IRPC-блока знак % рассматривается как обычный символ, а не как макрооператор.

Макрооператор ;;

Если в теле блока повторения (и макроса) имеются комментарии (в конце каких-то предложений или как самостоятельные предложения), то они переносятся во все копии блока. Однако бывает так, что комментарий полезен при описании самого блока повторения, но совершенно не нужен в его копиях. В таком случае следует начинать комментарий не с одной точки с запятой, а с двух – такие комментарии не копируются. Например:

```
IRP R,<AX,BX>
;;восстановление регистров          ;восстановить AX
;восстановить R                     POP AX
POP R  ;;стек ==> R           ⇒     ;восстановить BX
ENDM                                POP BX
```

11.3. Макросы

С помощью блока повторения один раз описывается некоторый фрагмент программы, который затем многократно копируется макрогенератором. Но блоки повторения можно использовать, только если эти копии должны быть расположены рядом друг с другом. А что делать, если фрагмент программы должен повторяться, но в разных местах программы? В этом случае используются макросы: специальным образом описывается этот фрагмент программы (это и есть макрос) и ему дается имя, а затем в нужных местах программы выписывается ссылка на этот макрос (указывается его имя); когда макроге-

нератор просматривает текст программы и встречает такую ссылку, то он вместо нее подставляет в окончательный текст программы сам макрос – соответствующий фрагмент программы; и так делается для каждой ссылки на макрос, в каком бы место программы она ни встретилась.

При использовании макросов применяется следующая терминология. Описание макроса называется макроопределением, ссылка на макрос – макрокомандой, процесс замены макрокоманды на макрос – макроподстановкой, а результат такой подстановки – макрорасширением.

11.3.1. Макроопределения

Описание макроса, т. е. макроопределение, имеет следующий вид:

```
<имя макроса> MACRO <формальные параметры через запятую>
              <тело макроса>
              ENDM
```

Два конкретных примера:

```
SUM MACRO X,Y ;X:=X+Y
    MOV AX,Y
    ADD X,AX
    ENDM
```

```
VAR MACRO NM,TP,VL
 NM D&TP VL
    ENDM
```

Первая строка макроопределения – это директива MACRO, которую принято называть заголовком макроса. В ней, во-первых, указывается имя, которое мы дали макросу, а во-вторых, через запятую перечисляются формальные параметры макроса. Необходимость в параметрах вызвана тем, что в общем случае макрос должен копироваться не в неизменном виде, а с некоторыми модификациями; параметры и обозначают те величины, которые влияют на эти модификации. Формальным параметрам можно давать любые имена, эти имена локализуются в теле макроса; если имя параметра совпало с именем другого объекта программы, то внутри макроопределения под этим именем понимается параметр, а не этот объект.

Тело макроса – это тот фрагмент программы, который затем и будет многократно копироваться. Тело может состоять из любого число любых предложений, в которых, естественно, можно использовать формальные параметры макроса. Как и в блоках повторения, формальные параметры могут обозначать любые части предложений тела. При этом, если рядом с параметром надо указать имя либо число или если параметр надо указать внутри строки, то следует использовать макрооператор & (см. D&TP в макросе VAR). В теле макроса можно использовать комментарии, начинающиеся с двух точек с запятой.

Завершает макроопределение директива ENDM (end of macro). Обратите особое внимание на то, что в этой директиве не надо повторять имя макроса (если здесь указать имя макроса, то это предложение будет рассматриваться как рекурсивный вызов макроса). Отметим также, что именно эта директива

указывается и в конце блоков повторения (в ЯА эти блоки рассматриваются как специфический случай макросов).

Где размещать макроопределения? Они могут быть размещены в любом месте текста программы (по ним в машинную программу ничего не записывается), но обязательно до первой ссылки на этот макрос. Таким образом, в ЯА действует правило: сначала опиши макрос и только затем обращайся к нему.

11.3.2. Макрокоманды

В тех местах программы, где мы хотим, чтобы макрогенератор подставил макрос, необходимо выписать обращения к макросу в виде макрокоманды, которая записывается следующим образом:

```
<имя макроса> <фактические параметры через запятую и/или пробел>
```

Конкретные примеры:

```
SUM A,ES:B      или    SUM A ES:B
VAR Z,W,?       или    VAR Z W,?
```

Как видно, макрокоманды очень похожи на обычные команды и директивы. Но есть и отличия. Во-первых, вместо названия команды или директивы, являющегося служебным словом, в макрокоманде указывается имя макроса, которое придумал сам автор программы. Во-вторых, в макрокоманде параметры могут отделяться друг от друга как запятыми, так и пробелами.

В качестве фактического параметра может быть указан любой текст (в том числе и пустой), но он должен быть сбалансирован по кавычкам и угловым скобкам и в нем не должно быть запятых, пробелов и точек с запятой вне этих кавычек и скобок. Поскольку запятыми и пробелами отделяется один параметр от другого, а с точки с запятой начинается комментарий, то их использование внутри фактического параметра привело бы к путанице.

При записи параметров макрокоманд можно использовать те же макрооператоры <>, ! и %, что и при записи фактических параметров блоков повторения. Например, если в фактическом параметре надо указать запятую, пробел или точку с запятой, то параметр следует заключить в уголки:

```
SUM <WORD PTR [SI]>,A
VAR C W <1,2>
```

При этом, напомним, уголки не считаются относящимися в параметру, а лишь указывают его границы.

Число фактических параметров, указываемых в макрокоманде, вообще говоря, должно равняться числу формальных параметров макроса, причем i-й фактический параметр соответствует i-му формальному параметру. Однако, если фактических параметров указано больше, чем надо, то лишние фактические параметры игнорируются, а если меньше, то считается, что в качестве недостающих фактических параметров заданы пустые тексты.

11.3.3. Макроподстановки и макрорасширения

Теперь рассмотрим, что происходит с макрокомандами. Когда макрогенератор, просматривая исходный текст программы, встречает макрокоманду, то он выполняет макроподстановку: находит описание макроса с указанным именем, берет его тело, заменяет в этом теле все формальные параметры на соответствующие фактические параметры и полученный таким образом текст (он называется макрорасширением) подставляет в программу вместо макрокоманды.

Примеры (над стрелкой указано, какие формальные параметры на какие фактические параметры заменяются):

```
                  X→A, Y→ES:B
SUM A,ES:B        ──────────►        MOV AX,ES:B
                                     ADD A,AX

                  NM→ , TP→W, VL→1,2
VAR ,W,<1,2>      ──────────────►    DW 1,2
```

В общем случае макрокоманда заменяется на несколько команд, т. е. она как бы представляет собой целую группу команд. Этим и объясняется название "макрокоманда" – "большая команда".

11.3.4. Примеры использования макросов

Пример 1 (описание крупных операций в виде макросов)

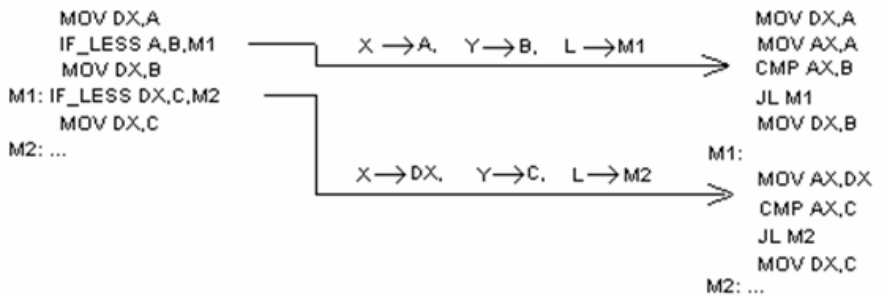
Одним из существенных недостатков машинного языка, который сохраняется и в ЯА, является то, что при программировании на этих языках приходится применять мелкие операции. Например, ЭВМ умеет складывать два числа, но вот три числа она сама по себе уже сложить не может, поэтому мы вынуждены указывать ей, как это делается. И такое сведение к мелким операциям приходится делать для любого алгоритма, сколь бы сложным он ни был. В этом и заключается основная сложность программирования на ЯА.

В определенной мере этот недостаток можно устранить с помощью макросов. Для этого надо в виде макросов описать более крупные операции, а затем составлять программу с использованием этих макросов.

Пусть, к примеру, в нашей программе многократно встречается условный переход "по меньше": if x<y then goto L. Эта операция реализуется тремя командами. Чтобы их каждый раз не выписывать заново, имеет смысл описать их как макрос, а затем пользоваться им. Давайте так и сделаем. Дадим этому макросу имя if_less и будем считать, что числа у нас знаковые и размером в слово:

```
IF_LESS MACRO X,Y,L
        MOV AX,X
        CMP AX,Y
        JL L
        ENDM
```

Имея такой макрос, можно, к примеру, следующим образом описать вычисление минимума трех чисел DX=min(A,B,C):



Слева указан текст программы, который составляет ее автор, а справа – те команды, которые будет реально выполняться. (Если макрокоманда помечена меткой, то в макрорасширении эта метка размещается в отдельной строке, а тело макроса начинается со следующей строки, т. к. в общем случае первая команда тела макроса может быть помечена своей меткой.)

Как видно, использование макросов сокращает размеры исходного текста программы и, что не менее важно, позволяет составлять программу в терминах более крупных операций. Если в виде макросов описать все часто используемые операции, то мы фактически построим новый язык, программировать на котором существенно проще, чем на "чистом" ЯА.

Пример 2 (макросы и обращения к процедурам)

Предположим, что имеется процедура NOD, вычисляющая наибольший общий делитель двух чисел: Z=NOD(X,Y), и что параметр X передается ее через регистр AX, параметр Y – через BX, а результат Z возвращается через AX. И пусть надо вычислить CX=NOD(A,B)+NOD(C,D). Тогда фрагмент программы, реализующий это вычисление, выглядит так:

```
MOV AX,A
MOV BX,B
CALL NOD
MOV CX,AX
MOV AX,C
MOV BX,D
CALL NOD
ADD CX,AX
```

Легко заметить, что при каждом обращении к процедуре приходится выписывать фактически одну и ту же группу команд. Так вот, чтобы их не выписывать каждый раз заново, можно эту группу команд описать как макрос, а затем его и использовать. Описание этого макроса выглядит так:

```
CALL_NOD MACRO X,Y
         MOV AX,X
         MOV BX,Y
         CALL NOD
         ENDM
```

а нужный нам фрагмент программы:

```
CALL_NOD A,B
MOV CX,AX
CALL_NOD C,D
ADD CX,AX
```

Как видно, получилось более наглядно и короче. После же макроподстановок получится тот же текст программы, что и прежде, но построением такого текста будет заниматься уже макрогенератор, а не мы.

Отметим попутно, что именно по этому принципу построены операции ввода-вывода, которыми мы пользуемся в данной книге. Например, ININT – это на самом деле макрос, описывающий обращение к процедуре, которая и реализует собственно ввод числа. (Описание этих макросов и процедур будет дано в гл. 13.)

Пример 3 (макросы и блоки повторения)

При входе в процедуру, как правило, требуется спасать в стеке содержимое регистров, для чего приходится выписывать несколько команд PUSH. И если в программе много процедур, если такая группа команд встречается многократно, то имеет смысл описать ее в виде макроса. Рассмотрим, как выглядит такой макрос.

Прежде всего отметим, что по смыслу у этого макроса может быть любое число фактических параметров (разное число названий регистров). Поскольку в ЯА можно определять макросы только с фиксированным числом формальных параметров, то в подобных ситуациях обычно поступают так: макрос описывают с одним формальным параметром, но при обращении к нему (в макрокоманде) указывают через запятую нужное число фактических параметров и заключают весь их список в угловые скобки, в результате чего синтаксически получается один параметр. В теле же макроса от этого списка "отщепляют" по одному настоящему параметру и что-то с ним делают.

Далее. Макрос должен поставлять в текст окончательной программы (в макрорасширение) несколько однотипных команд PUSH r. Ясно, что здесь следует воспользоваться блоком повторения.

С учетом всего сказанного получаем такое определение нашего макроса:

```
SAVE MACRO REGS ;;запись в стек регистров из списка REGS
     IRP R,<REGS>
     PUSH R
     ENDM
     ENDM
```

Рассмотрим, как макрогенератор обрабатывает следующее обращение к нашему макросу:

```
SAVE <AX,SI,BP>
```

Прежде всего, как обычно, выписывается тело макроса с заменой формального параметра на фактический:

```
IRP R,<AX,SI,BP>
PUSH R
ENDM
```

Напомним, что уголки, в которые был заключен фактический параметр, считаются не относящимися к параметру и при макроподстановке удаляются. Однако в директиве IRP были свои уголки, именно они-то и сохранились в тексте, так что эта директива сохранила правильный синтаксис.

Получившийся текст содержит конструкцию макроязыка, а в такой ситуации, как уже отмечалось, макрогенератор продолжает свою работу. В данном случае он "раскручивает" IRP-блок, в результате чего получает уже текст на "чистом" ЯА:

```
PUSH AX
PUSH SI
PUSH BP
```

который и является окончательным макрорасширением нашей макрокоманды.

11.3.5. Макросы и процедуры

Прежде чем продолжить рассказ про макросы, сравним их с процедурами.

В чем выгода от макросов? Если у нас в тексте программы многократно встречается один и тот же (или почти один и тот же) фрагмент, то чтобы не выписывать его каждый раз заново, мы описываем его как макрос, а затем в нужных местах программы указываем лишь короткие макрокоманды, чем существенно облегчаем себе жизнь. Таким образом, макросы – это средство борьбы с повторяющимися действиями. Но процедуры – это тоже средство борьбы с повторяющимися действиями: многократно повторяющуюся группу команд мы можем описать как процедуру, а затем в нужным местах программы будем указывать команды вызова процедуры.

Итак, имеется два средства решения одной и той же проблемы. Естественно, возникает вопрос: какое из этих средств лучше? Чтобы ответить на этот вопрос, надо понять различие между макросами и процедурами. А оно проявляется в следующем.

И макрос, и процедура описываются в программе только 1 раз. В обоих случаях в нужных местах программы мы указываем короткие ссылки на это описание. Таким образом, с точки зрения процесса написания текста программы, особого различия между макросами и процедурами нет. Но вот после того, как мы написали программу, дальнейшая судьба макросов и процедур оказывается совершенно различной.

В оттранслированной программе процедура так и остается в единственном экземпляре, а при использовании макроса его тело подставляется во все места, где мы указали обращение к нему, т. е. макрос "размножается," и по-

тому размеры программы увеличиваются. Таким образом, применение процедур дает выигрыш по памяти, и в этом их преимущество перед макросами.

Однако процедуры проигрывают по времени. В самом деле, при обращении к процедуре мы должны выполнить команды засылки ее параметров в те или иные регистры (либо в стек) и выполнить команду вызова процедуры. Кроме того, по окончании работы процедуры мы выполняем команду возврата из нее. На выполнение всех этих команд, естественно, затрачивается время. Если же используется макрос, то все эти команды выполнять не надо. Действительно, поскольку команды макроса подставляются в то место программы, где их надо выполнить, то команды перехода не нужны. Не нужны и команды передачи параметров, поскольку настройка на них происходит при макроподстановке, когда в теле макроса формальные параметры заменяются на фактические. Тем самым при использовании макроса мы не тратим время на переходы и передачу параметров, а поэтому и получаем выигрыш во времени.

Конечно, при замене макрокоманд на макрорасширения тратится время, но это происходит еще на этапе трансляции, до выполнения программы. Когда же программа будет оттранслирована и начнет выполняться, то здесь уже не расходуется время на переходы и передачу параметров. При использовании же процедур время на эти действия тратится как раз на этапе выполнения программы. А именно время счета программы, а не время ее трансляции, определяет эффективность программы.

Итак, процедуры дают выигрыш по памяти, а макросы – выигрыш по времени. Поэтому нельзя сказать, что процедуры лучше макросов или наоборот. Каждое из этих средств хорошо в определенных условиях, в одних случаях лучше процедуры, в других – макросы. Если в повторяющемся фрагменте программы много команд (десяток и более), то лучше описать его как процедуру. В самом деле, если этот фрагмент описать как макрос, то он затем будет многократно размножен и программа сильно "разбухнет", она будет занимать много места в памяти. В то же время процедура всегда хранится в одном экземпляре и программа не "разбухает". Но если в повторяющемся фрагменте мало команд, тогда его лучше описать как макрос. Конечно, и здесь произойдет "разбухание" программы, но не так уж и сильно. В то же время, если небольшую группу команд описать как процедуру, то число вспомогательных команд по вызову ее и передаче ей параметров станет сравнимым с числом команд самой процедуры, поэтому время ее выполнения станет чуть ли не вдвое больше времени выполнения макроса.

Итак, большие фрагменты рекомендуется описывать как процедуры, а маленькие – как макросы.

11.3.6. Определение макроса через макрос

Как известно, процедура имеет право обращаться к другой процедуре. Аналогично, при описании одного макроса можно ссылаться на другой мак-

рос. В частности, допускается и обращение макроса к самому себе, т. е. разрешены рекурсивные макросы. Однако рекурсивные макросы встречаются на практике крайне редко, поэтому рассмотрим лишь пример нерекурсивного обращения одного макроса к другому и возникающие при этом проблемы.

Пусть макрос ARR X,N предназначен для описания массива X из N байт:

```
ARR MACRO X,N
  X DB N DUP(?)
    ENDM
```

Тогда, используя его, можно определить макрос ARR2, предназначенный для описания сразу двух массивов одного и того же размера;

```
ARR2 MACRO X1,X2,K
     ARR X1,<K>
     ARR X2,<K>
     ENDM
```

При таком макроопределении макроподстановка для макроса ARR2 проходит в два этапа, например:

```
ARR2 A,B,20 ⇒    ARR A,<20>  ⇒  A DB 20 DUP(?)
                 ARR B,<20>     B DB 20 DUP(?)
```

Почему в теле макроса ARR2 при обращении в макросу ARR второй фактический параметр записывается в уголках? Дело в том, что если по смыслу первым и вторым параметрами макроса ARR2 могут быть только имена, то как третий параметр может быть указана достаточно сложная конструкция, например: ARR2 A,B,<25 MOD 10>. Так вот, если бы вместо записи <K> использовалась просто запись K, тогда на первом этапе макроподстановки получилась бы макрокоманда ARR A,25 MOD 10 с четырьмя операндами, а не с двумя (напомним, что при макроподстановке уголки фактического параметра отбрасываются и что в макрокомандах параметры могут отделяться как запятыми, так и пробелами). При записи же <K> уголки заставляют рассматривать эту конструкцию как один параметр: ARR A,<25 MOD 10>.

Отметим, что в ЯА допускается вложенность макроопределений, например:

```
ARR2 MACRO X1,X2,K
 ARR MACRO X,N
   X DB N DUP(?)
     ENDM
     ARR X1,<K>
     ARR X2,<K>
     ENDM
```

Однако при этом надо учитывать следующее. Макрос ARR, хотя и описан внутри макроса ARR2, не локализуется в ARR2, и к нему можно обращаться вне макроса ARR2. Но ассемблер работает так, что описание внутреннего

макроса он "замечает" только при первом обращении к внешнему макросу. Поэтому обращаться к макросу ARR до обращения к макросу ARR2 нельзя:

```
ARR A,50        ;ошибка (имя ARR еще не описано)
ARR2 B,C,100
ARR D,60        ;можно (имя ARR уже описано)
```

11.3.7. Директива LOCAL

При использовании макросов возникает неприятная проблема с метками, которыми могут быть помечены предложения тела макроса. Пусть, к примеру, имеется макрос со следующей структурой:

```
 M MACRO
    ...
L:  ...
    ...
   ENDM
```

и пусть в программе имеется два обращения к этому макросу. Тогда после макроподстановок мы получим следующую картину:

```
                    ...
M       ⇒       L:  ...
...                 ...
M       ⇒       L:  ...
                    ...
```

Как видно, в окончательном тексте программы появились две команды, помеченные одной и той же меткой L, а это ошибка. Почему так произошло? Дело в том, что имя L не является формальным параметром макроса и потому при макроподстановке ни на что не заменяется, а переносится в макрорасширение без всяких изменений. Вот и получается, что в окончательном тексте программы будет столько меток L, сколько было обращений к данному макросу.

Как избежать этой ошибки? Возможный вариант – включить метку L в число параметров макроса и при обращении к макросу указывать различные фактические метки. Однако это плохое решение, поскольку метки, которыми метятся команды тела макроса, – это чисто внутреннее дело макроса, и для того, кто будет пользоваться этим макросом, нет никакого дела до этих меток, придумывать имена для таких внутренних меток – это лишняя работа.

Учитывая это, в ЯА предложено иное, более удобное решение данной проблемы. Оно заключается в том, что после заголовка макроса (директивы MACRO) надо указать специальную директиву макроязыка:

```
LOCAL v1, ..., vk  (k>=1)
```

где vi – имена, используемые в макроопределении (обычно это метки). Тогда при макроподстановке макрогенератор будет заменять эти имена на специальные имена вида ??xxxx, где xxxx – четырехзначное шестнадцатеричное число, т. е. на имена ??0000, ??0001 и так далее до ??FFFF. Правила такой замены следующие.

Макрогенератор запоминает номер, который он использовал в последний раз; пусть это был номер n. Когда макрогенератор встречает обращение к макросу, в котором имеется директива LOCAL, то он ставит в соответствие именам, перечисленным в этой директиве, специмена с очередными номерами: специмя ??(n+1) для v1, специмя ??(n+2) для v2 и т. д., а затем при макроподстановке заменяет каждое вхождение vi на одно и то же специмя ??(n+i). Когда макрогенератор встретит новое обращение к этому же или другому макросу, где есть директива LOCAL, то он будет уже использовать специмена с последующими номерами: n+k+1 и т. д. Поэтому в разных макрорасширениях появятся разные специмена, совпадений не будет.

Рассмотрим конкретный пример. Опишем макрос, вычисляющий остаток от деления одного натурального числа на другое с помощью вычитания:

```
MD MACRO R1,R2  ;r1:=r1 mod r2 (r1,r2 – регистры, без знака)
   LOCAL M,M1
M: CMP R1,R2    ;;while r1>=r2 do r1:=r1-r2
   JB M1
   SUB R1,R2
   JMP M
M1:
   ENDM
```

и предположим, что в последний раз макрогенератор использовал для специмен номер 0016. Тогда два очередных обращения к макросу MD будут обработаны следующим образом:

```
            R1→AX, R2→BX                ??0017: CMP AX,BX
MD AX,BX    ------------>                        JB ??0018
            M→??0017, M1→??0018                 SUB AX,BX
                                                JMP ??0017
                                        ??0018:
          R1→CX, R2→DX                  ??0019: CMP CX,DX
MD CX,DX  ------------>                         JB ??001A
          M→??0019, M1→??001A                   SUB CX,DX
                                                JMP ??0019
                                        ??001A:
```

Таким образом, в разных макрорасширениях появляются разные метки, и тем самым проблема с одинаковыми метками снимается.

Отметим, что директиву LOCAL можно указывать только после директивы MACRO (но зато любое число раз) и нигде более и что директива LOCAL не переносится в макрорасширение. Кроме того, следует учитывать, что специмена ??0017 и т. п. – это обычные имена, и, вообще говоря, мы можем их использовать сами, однако в силу их особой роли не рекомендуется применять эти имена в ином качестве.

11.3.8. Директива EXITM

Рассмотрим еще одну директиву макроязыка:

```
EXITM
```

У этой директивы нет операндов. Ее можно использовать только внутри макроопределений и блоков повторения, т. е. внутри конструкций макроязыка, оканчивающихся директивой ENDM. Встретив директиву EXITM, макрогенератор завершает обработку ближайшего объемлющего макроопределения или блока повторения.

Например, при макроопределении

```
A MACRO K
  REPT K
  DB 0
  EXITM
  ENDM
  DW ?
  ENDM
```

макрокоманда A 5 будет заменена на следующий текст:

```
DB 0
DW ?
```

Здесь макрогенератор, создавая первую копию тела блока повторения, перенесет предложение DB 0 в макрорасширение, а затем, встретив EXITM, полностью завершит обработку этого блока, но не покинет тело макроса – он "перескочит" за ближайшую директиву ENDM, т. е. на предложение DW ?.

Более содержательные примеры на директиву EXITM будут приведены позже (они требуют знания условных директив). Пока лишь отметим, что эта директива используется тогда, когда при выполнении некоторого условия надо досрочно (не доходя до ENDM) прекратить макроподстановку или "раскрутку" блока повторения.

11.3.9. Переопределение и отмена макросов

В отличие от других объектов программы на ЯА, макросы можно переопределить или уничтожить.

Если в тексте программы описать макрос с именем, которым ранее был обозначен другой макрос, то с этого момента прежний макрос считается уничтоженным, а новый макрос – действующим. Например:

```
A MACRO X
  INC X
  ENDM
  A CX       ⇒   INC CX

A MACRO Y,Z
  CMP Y,0
  JE Z
  ENDM
  A BH,EQ    ⇒   CMP BH,0
                 JE EQ
```

Макрос можно и просто уничтожить, не определяя новый макрос с тем же именем, для чего надо воспользоваться следующей директивой:

```
PURGE <имя макроса> {,<имя макроса>}
```

После этой директивы все макросы, имена которых в ней перечислены, считаются несуществующими. Например, после директивы

```
PURGE A,ININT
```

к макросам A и ININT уже нельзя обращаться.

11.4. Условное ассемблирование

Если макросы и блоки повторения позволяют избежать многократного выписывания в исходном тексте программы повторяющихся фрагментов, то рассматриваемое в данном разделе средство макроязыка – условное ассемблирование – удобно при многократных прогонах программы. Оно дает возможность в исходном тексте держать несколько вариантов одного и того же участка программы, а при каждом ее прогоне оставлять в окончательном тексте только один из них. Какой именно вариант будет оставлен, зависит от тех или иных условий, которые автор программы задает перед прогоном. Выгода от такой условной сборки окончательного текста программы заключается в том, что автор программы не должен перед каждым ее прогоном вручную редактировать ее текст (это чревато ошибками и требует значительного времени), а возлагает эту работу на макрогенератор (он работает без ошибок и быстрее).

Участок программы, затрагиваемый условным ассемблированием, должен записываться в виде так называемого IF-блока:

```
<IF-директива>          <IF-директива>
<фрагмент-1>     или     <фрагмент-1>
ELSE                     ENDIF
<фрагмент-2>
ENDIF
```

Директивы ELSE и ENDIF обязательно должны записываться в отдельных строчках. В каждом же фрагменте может быть любое число любых предложений, в частности в них снова могут быть IF-блоки, т. е. допускается вложенность IF-блоков.

В IF-директиве (имеется несколько разновидностей ее) указывается некоторое условие, которое проверяется макрогенератором. Если условие выполнено, то макрогенератор оставляет в окончательном тексте программы только фрагмент-1, а фрагмент-2 исключает, не переносит в окончательный текст. Если же условие не выполнено, тогда фрагмент-1 игнорируется, а в окончательную программу вставляется только фрагмент-2. Если части ELSE нет, то считается, что фрагмент-2 пуст, поэтому при невыполнении условия такой IF-блок ничего не "поставляет" в окончательный текст программы.

Поскольку условие в IF-директиве проверяется на этапе макрогенерации, то, вполне естественно, в нем не должно быть ссылок на величины, которые станут известными только при выполнении программы (например, в условии

нельзя ссылаться на содержимое регистров или ячеек памяти). Более того, условие должно быть таким, чтобы макрогенератор мог вычислить его сразу, как только встретит его (например, в нем не должно быть ссылок вперед).

В макроязыке довольно-таки много разновидностей IF-директивы. Мы будем рассматривать их парами, в каждой из которых директивы проверяют противоположные условия.

11.4.1. Директивы IF и IFE

Эти директивы имеют следующий вид:

```
IF <константное выражение>
IFE <константное выражение>
```

Встречая любую из этих директив, макрогенератор вычисляет указанное к ней константное выражение. В директиве IF условие считается выполненным, если значение выражения отлично от 0, а директиве IFE (if equal, если равно) – если значение равно 0.

Рассмотрим следующий пример. Предположим, что мы отлаживаем программу и для этого вставляем в определенные места ее текста отладочные печати (ОП), т. е. печать промежуточных значений каких-то переменных. Закончив отладку, мы, естественно, убираем ОП из текста. Но может оказаться так, и это действительно часто случается, что затем в программе опять появятся ошибки, и чтобы их найти, нам придется снова вставить те же ОП, а после исправления ошибок – снова удалить. И этот процесс вставки и удаления ОП может продолжаться долго.

Вставлять и убирать ОП мы можем, конечно, сами, с помощью какого-нибудь текстового редактора. Но если ОП много, если они разбросаны по всей программе, то такие изменения текста займут много времени, а кроме того, здесь легко и ошибиться – мы можем не туда вставить ОП или можем убрать из текста не то, что надо. Так вот, в подобной ситуации как раз и удобно использовать возможности условного ассемблирования: в тексте программы постоянно сохраняем ОП, но перед каждой из них указываем условие, что команды ОП должны оставаться в окончательном тексте программы только при отладке.

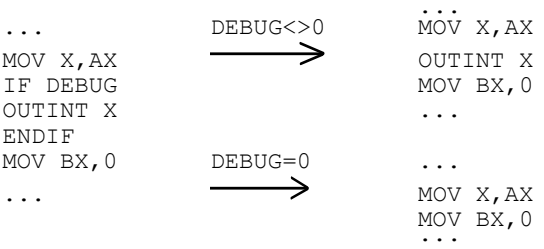
Конкретно это можно сделать так. Договоримся, что режим прогона программы (отладка или счет) указывается с помощью константы DEBUG (отладка), которую описываем в начале текста программы и которой присваиваем значение 1, если у нас сейчас отладка:

```
DEBUG EQU 1
```

или значение 0 при счете:

```
DEBUG EQU 0
```

Тогда участок исходной программы с отладочной печатью (скажем, переменной X) должен быть записан так, как указано слева, в окончательном же тексте программы этот участок будет выглядеть так, как изображено справа:



При таком построении текста программы нам достаточно перед ее прогоном менять лишь одну строчку – описание константы DEBUG, чтобы макрогенератор формировал разный окончательный текст программы (с ОП или без них). Менять что-либо еще в тексте программы нам уже не надо.

Здесь следует обратить внимание на такую вещь. Сделать так, чтобы ОП выполнялись при отладке и не выполнялись при счете, можно и с помощью команд условного перехода. Выглядит это примерно так:

```
   ...
   MOV X,AX
   CMP отладка?
   JNE L
   OUTINT X
L: MOV BX,0
   ...
```

Казалось бы, получили то же самое решение проблемы. Однако это далеко не так. В чем разница? В том, что в последнем случае команды ОП остаются в программе всегда, нужны они или нет, и потому будут всегда занимать место в памяти. Кроме того, здесь проверка, выполнять команды ОП или нет, делается в процессе выполнения программы, и на это тратится время. При условном же ассемблировании команды ОП, если они не нужны, будут удалены из программы и потому не будут занимать место в памяти. Кроме того, здесь проверка, оставлять команды ОП или нет, осуществляется еще до того, как программа начнет выполняться, и потому в процессе выполнения программы никаких проверок и обходов уже не будет. Таким образом, при условном ассемблировании затрачивается больше времени на трансляцию программы, зато сама программа получится более экономной и по памяти, и по времени. В этом и заключается преимущество условного ассемблирования перед использованием команд условного перехода.

Рассмотрим еще один пример на использование директив IF и IFE, в котором средства условного ассемблирования используются не самостоятельно, как в предыдущем примере, а для описания макросов, что является основным случаем применения этих средств.

Опишем в виде макроса SHIFT X,N операцию сдвига X на N разрядов вправо при условии, что параметр X – это имя какой-то переменной, а N – явно заданное положительное число, и при условии, что макрорасширение этого макроса должно содержать минимально возможное число команд.

Последнее условие означает, что по макрокоманде SHIFT X,N при N=1 должно формироваться макрорасширение из одной команды (SHR X,1),

а при N>1 – из двух команд (MOV CL,N и SHR X,CL). Поскольку здесь в зависимости от некоторого условия (величины N) в окончательный текст программы (в макрорасширение) должны попадать разные фрагменты, то ясно, что при описании макроса следует воспользоваться средствами условного ассемблирования. Сделать это можно, например, так:

```
SHIFT MACRO X,N
      IFE N-1      ;;N-1=0? (N=1?)
      SHR X,1
      ELSE         ;;N>1
      MOV CL,N
      SHR X,CL
      ENDIF
      ENDM
```

Ниже для примера показано, как осуществляется макроподстановка для макрокоманды SHIFT A,5:

```
            X→A, N→5   IFE 5-1      5-1=0?   MOV CL,5
SHIFT A,5   ─────────→   SHR A,1     ──────→   SHR A,CL
                       ELSE         нет
                        MOV CL,5
                        SHR A,CL
                       ENDIF
```

11.4.2. Операторы отношения. Логические операторы

В общем случае константное выражение, указываемое в директивах IF и IFE, может быть любым, но по смыслу оно, конечно, должно быть логическим выражением. До сих пор мы рассматривали только арифметические (числовые) выраже-ния ЯА, а теперь рассмотрим, как записываются логические выражения. Они считаются частным случаем константных выражений "вообще", поэтому, хотя в основном используются в конструкциях макроязыка, принадлежат все же "чистому" ЯА.

Прежде всего отметим, что в ЯА "ложью" считается число 0, а "истиной" считается число -1 (0FFFFh).

В ЯА имеется шесть операторов отношения:

```
<выражение> EQ <выражение>
<выражение> NE <выражение>
<выражение> LT <выражение>
<выражение> LE <выражение>
<выражение> GT <выражение>
<выражение> GE <выражение>
```

Названия этих операторов расшифровываются следующим образом: EQ (equal) - равно, NE (not equal) – не равно, LT (less than) – меньше, LE (less or equal) – меньше или равно, GT (greater than) – больше, GE (greater or equal) – больше или равно. Оба операнда должны быть либо константными выражениями, либо адресными выражениями, значениями которых обязаны быть адреса из одного и того же сегмента памяти. Если проверяемое отношение

выполняется, то значением оператора является "истина", не выполняется – "ложь". Например, если константа N имеет значение 5, тогда:

```
N+1 LT 5 → 0 (ложь)
N-1 LT 5 → -1 (истина)
```

Следует, однако, отметить, что действия этих операторов слегка "заумны". Операторы EQ и NE трактуют свои операнды как 16-битовые знаковые числа в дополнительном коде, поэтому выражение -2 EQ 0FFFEh оказывается истинным. Остальные же операторы отношения правильно учитывают знаки своих операндов (например, отношение -1 LT 0FFFEh истинно), кроме одного случая: если сравниваемые операнды равны как 16-битовые знаковые числа в дополнительном коде, то эти операторы считают их равными (например, отношение -1 LE 0FFFFh считается ложным). Учитывая подобные "фокусы", лучше не использовать отрицательные числа в этих операторах.

В качестве примера на использование операторов отношения опишем в виде макроса SET0 X действие X:=0 при условии, что X – имя переменной размером в байт, слово или двойное слово:

```
SET0 MACRO X
     IF TYPE X EQ DWORD     ;;двойное слово (тип X равен 4)?
     MOV WORD PTR X,0
     MOV WORD PTR X+2,0
     ELSE                   ;;байт или слово
     MOV X,0
     ENDIF
     ENDM
```

Логические значения и отношения можно объединять в более сложные логические выражения с помощью следующих логических операторов:

```
NOT <константное выражение>
<константное выражение> AND <константное выражение>
<константное выражение> OR <константное выражение>
<константное выражение> XOR <константное выражение>
```

Эти операторы реализуют соответственно операции отрицания, конъюнкции, дизъюнкции и "исключающего ИЛИ".

В качестве примера опишем в виде макроса RSH B,N сдвиг значения байтовой переменной B на N разрядов вправо при условии, что N – явно заданное неотрицательное число. Мы не будем выделять особо случай N=1, когда сдвиг можно осуществить одной командой, однако учтем, что при N=0 сдвиг не нужен (макрорасширение должно быть пустым), а при N>7 результат сдвига известен заранее (это 0), поэтому сдвиг можно заменить записью нуля в B. С учетом этого получаем такое макроопределение:

```
RSH MACRO B,N
    IF (N GT 0) AND (N LT 8)  ;; 0<N<8 ?
    MOV CL,N
    SHR B,CL
    ELSE
```

```
    IF N GE 8                   ;; N>=8 ?
    MOV B,0
    ENDIF
    ENDIF
    ENDM
```

Теперь уточним действия логических операторов. По смыслу их операндами, конечно, должны быть логические выражения, однако в ЯА ими могут быть и любые другие константные выражения (но не адресные), значения которых трактуются как 16-битовые слова. Значением этих операторов также является 16-битовое слово, которое получается в результате поразрядного выполнения соответствующей операции (i-й бит результата определяется только i-ми битами операндов). Например:

```
SCALE EQU 1010b
      MOV AX,SCALE AND 11b     ;эквивалентно MOV AX,10b
      AND AX,SCALE XOR 11b     ;эквивалентно AND AX,1001b
```

Другими словами, эти операторы выполняются аналогично одноименным командам ПК. Однако не следует путать эти операторы и команды: операторы используются для записи операндов команд и директив и вычисляются еще на этапе трансляции программы (в машинной программе их уже нет), а команды выполняются на этапе счета программы.

11.4.3. Директивы IFIDN, IFDIF, IFB и IFNB

Вернемся к IF-директивам и следующей рассмотрим такую пару директив:

```
IFIDN <t1>,<t2>
IFDIF <t1>,<t2>
```

Здесь t1 и t2 – любые тексты (последовательности символов), причем они обязательно должны быть заключены в уголки. Эти тексты посимвольно сравниваются. В директиве IFIDN условие считается выполненным, если эти тексты равны (идентичны, identical), а в директиве IFDIF – если они не равны (различны, different). Отметим, что при сравнении этих текстов большие и малые буквы не отождествляются.

Примеры:

```
IFIDN <a+b>,<a+b>  - условие выполнено
IFIDN <a+b>,<a>    - условие не выполнено
IFIDN <a+b>,<a+B>  - условие не выполнено
```

Обе эти директивы имеет смысл использовать лишь внутри тела макроса (или блока повторения), указывая в сравниваемых текстах формальные параметры макроса. При макроподстановке эти параметры будут заменяться на фактические параметры, и это позволяет проверить, заданы ли при обращении к макросу фактические параметры определенного вида или нет.

В качестве примера опишем в виде макроса MM R1,R2,T (где R1 и R2 – имена байтовых регистров, содержимое которых трактуется как знаковые числа, а T – слово MAX или MIN) операцию R1:=T(R1,R2), т. е. запись в R1 либо максимума чисел R1 и R2 (при T=MAX), либо минимума.

Прежде всего отметим, что этот макрос должен генерировать непустое макрорасширение, только если R1 и R2 – разные регистры (при обращении же, скажем, MM AL,AL,MAX менять значение AL не надо). Как проверить несовпадение двух первых фактических параметров макроса? Отметим, что директива IF R1 NE R2 здесь не проходит, т. к. оператор NE предназначен для сравнения чисел, а нам надо сравнить имена регистров. Зато здесь проходит директива IFDIF <R1>,<R2>, в которой при макроподстановке формальные параметры R1 и R2 будут заменены на имена регистров (для указанной выше макрокоманды эта директива примет вид IFIDN <AL>,<AL>), что и позволит сравнить эти имена.

Далее. Чтобы определить, что именно надо вычислять – максимум или минимум, надо проверить третий фактический параметр: MAX это или MIN? Сделать это можно, например, с помощью директивы IFIDN <T>,<MAX>: при макроподстановке формальный параметр T будет заменен на третий фактический параметр, который тем самым и будет сравниваться со словом MAX.

И последнее. Вычисления максимума и минимума различаются лишь одной командой – командой условного перехода:

```
;R1:=max(R1,R2)      ;R1:=min(R1,R2)
   CMP R1,R2            CMP R1,R2
   JGE L                JLE L
   MOV R1,R2            MOV R1,R2
L:                   L:
```

поэтому не имеет смысла в макроопределении дважды выписывать эту практически одинаковую группу команд, а достаточно выписать ее лишь раз, поместив вместо команды условного перехода IF-блок, который в зависимости от параметра T выберет нужную команду условного перехода.

С учетом всего сказанного получаем такое описание макроса MM:

```
MM MACRO R1,R2,T
   LOCAL L
   IFDIF <R1>,<R2>    ;;R1 и R2 – разные регистры?
   CMP R1,R2
   IFIDN <T>,<MAX>    ;;T=MAX?
   JGE L              ;;да – поместить JGE L в макрорасширение
   ELSE
   JLE L              ;;нет – поместить JLE L
   ENDIF
   MOV R1,R2
L:
   ENDIF
   ENDM
```

Приведем этапы макроподстановки для макрокоманды MM AL,BH,MIN после того, как в теле макроса формальные параметры были заменены на фактические параметры, а локальная метка L – на специмя (скажем, ??0105):

```
   IFDIF <AL>,<BH>
   CMP AL,BH                CMP AL,BH                  CMP AL,BH
   IFIDN <MIN>,<MAX>        IFIDN <MIN>,<MAX>          JLE ??0105
   JGE ??0105          ⇒    JGE ??0105          ⇒      MOV AL,BH
   ELSE                     ELSE                    ??0105:
   JLE ??0105               JLE ??0105
   ENDIF                    ENDIF
   MOV AL,BH                MOV AL,BH
??0105:                  ??0105:
  ENDIF
```

Как уже отмечалось, при сравнении текстов в директивах IFIDN и IFDIF большие и малые буквы не отождествляются. Это неприятная особенность данных директив. Обычно с их помощью в макросах сравниваются имена переменных и регистров, а эти имена в текстах программ могут записываться как малыми, так и большими буквами или вперемежку, и если при обращении к макросу имена записаны не теми буквами, как предусмотрел автор макроса, то этот макрос может сформировать неправильное макрорасширение. Например, по макрокоманде MM AL,AH,Max будет построена группа команд для вычисления минимума (!) чисел из регистров AL и AH, т. к. в теле макроса MM директива IFIDN будет сравнивать тексты Max и MAX, которые не совпадают.

Как преодолеть этот недостаток? Можно предложить следующий метод: берем все возможные варианты записи имени (например, AH, Ah, aH и ah) и с помощью IRP-блока и директивы IFIDN проверяем, не совпадает ли заданное имя с одним из этих вариантов; если совпадает, тогда некоторому вспомогательному имени с помощью директивы присваивания (=) даем одно значение, не совпадает – другое; затем это значение нужно проверить директивой IF.

Эти действия (кроме последней проверки) можно описать в виде следующего макроса SAME N,LN,F (где N – то, что проверяем; LN – список (в уголках, через запятую) всех возможных вариантов записи имени, для которого делаем проверку; F – имя, которому присваивается результат проверки: -1 (истина) – есть совпадение, 0 – нет):

```
SAME MACRO N,LN,F
     F=0             ;;присвоить 0 имени F
                     ;;(на случай несовпадения)
     IRP V,<LN>      ;;для каждого варианта V из LN выполнить:
     IFIDN <V>,<N>   ;;если V совпал с N, то
     F=-1            ;;переприсвоить F значение -1
     EXITM           ;;прекратить макроподстановку
     ENDIF
     ENDM
     ENDM
```

Хотя макроподстановка для этого макроса может выполняться достаточно долго, в конце концов генерируется либо одна, либо две директивы присваивания, например:

```
SAME Ah,<AH,Ah,aH,ah>,AX? ⇒ AX?=0
                            AX?=-1

SAME bh,<AH,Ah,aH,ah>,AX? ⇒ AX?=0
```

Используем SAME для описания в виде макроса CALL_P X команд обращения к процедуре P, параметр (X) для которой передается через регистр AX.

В общем случае этот макрос должен обозначать следующие команды:

```
MOV AX,X
CALL P
```

Однако, если в качестве X указан регистр AX, то первая команда не нужна. Таким образом, здесь надо сравнивать X с именем AX, причем с любым из возможных вариантов его записи. Для этого мы и воспользуемся макросом SAME, причем имя, которому присваивается результат проверки, локализуем в макросе CALL_P:

```
CALL_P MACRO X
       LOCAL F
       SAME X,<AX,Ax,aX,ax>,F   ;;F:="истина", если X - это AX
       IFE F                    ;;если F - "ложь", то
       MOV AX,X                 ;;сформировать команду AX:=X
       ENDIF
       CALL P
       ENDM
```

Теперь рассмотрим еще две IF-директивы:

```
IFB <t>
IFNB <t>
```

Они фактически являются вариантами директив IFIDN и IFDIF, когда второй текст пуст. В директиве IFB (if blank, если пусто) условие считается выполненным, если текст t пуст, а в директиве IFNB (if not blank) - если текст t не пуст.

Эти директивы используются в макросах для проверки, задан ли фактический параметр или нет (не задать параметр – значит указать в соответствующей позиции макрокоманды пустой текст). Например, макроопределение

```
DEF MACRO X,V
    IFB <V>  ;;параметр V не задан (пустой)?
  X DB ?
    ELSE
  X DB V
    ENDIF
    ENDM
```

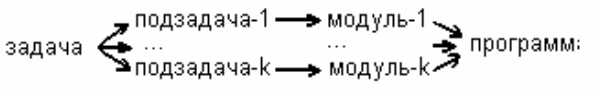
описывает макрос DEF X,N, по которому определяется байтовая переменная X с начальным значением V, если оно указано, или без него:

```
DEF A,6     ⇒    A DB 6
DEF B       ⇒    B DB ?
```

Помимо рассмотренных здесь IF-директив имеются и другие IF-директивы, но на практике они используются не очень часто, поэтому будут описаны в гл. 14, где собраны редко встречающиеся конструкции ЯА.

12. МНОГОМОДУЛЬНЫЕ ПРОГРАММЫ

Хорошо известен следующий прием, упрощающий составление больших программ: если решаемая задача сложна и сразу написать программу ее решения трудно, тогда следует разделить задачу на несколько более простых подзадач, для каждой из них написать соответствующую часть программы, а затем объединить эти части в единую программу:



Термином "модуль" принято называть часть программы, решающую некоторую подзадачу и более или менее независимую от других частей.

Выгода от указанного способа составления программы не только в том, что подзадачи проще исходной задачи и потому написать модули легче, чем всю программу целиком, но и в том, что разработку модулей можно поручить разным людям, которые могут работать параллельно, а это позволяет существенно сократить время разработки программы в целом.

Частным случаем модулей являются процедуры. Здесь в качестве модулей выделяются многократно встречающиеся части программы. Однако программу можно разделить и на такие модули, которые выполняются всего один раз. На какие конкретно модули делить программу, решает автор программы, это его личное дело.

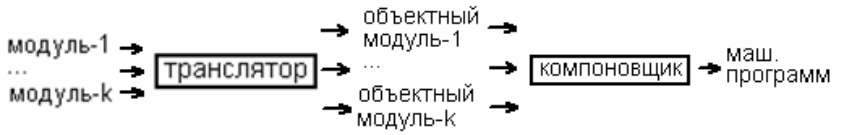
Существуют два основных варианта объединения модулей. В первом варианте модули объединяются в единую программу до того, как начнется трансляция программы:

Именно этот вариант обычно подразумевается при использовании процедур: процедуры и основную часть программы мы можем составлять и по отдельности, но затем мы их объединяем вместе и только после этого транслируем программу. Аналогичный способ объединения дает и применение директивы INCLUDE, по которой, напомним, содержимое указанного файла вставляется в текст программы, причем эта вставка делается до перевода программы на машинный язык.

Этот способ объединения модулей является основным для небольших программ, с небольшим числом модулей. Но если в программе много модулей, то у этого способа проявляется недостаток, суть которого в следующем: если программа разделена на несколько модулей и в одном из них обнаружена ошибка, то после ее исправления мы вынуждены будем оттранслировать

заново всю программу, все ее модули. Это и плохо: изменили только один модуль, а перетранслировать приходится и все остальные модули, что ведет к большим потерям времени.

В другом способе модули транслируются независимо друг от друга и лишь после этого происходит их объединение:



Объектным модулем принято называть модуль в оттранслированном виде, в машинных кодах. Объединением таких модулей в единую машинную программу занимается специальная программа, называемая компоновщиком (linker).

В данном случае не надо перетранслировать все модули программы, если изменился лишь один модуль, достаточно перетранслировать только этот модуль и затем снова объединить его с другими, ранее оттранслированными модулями. Конечно, на объединение объектных модулей уходит время, но оно существенно меньше времени трансляции модулей.

Именно этот второй вариант объединения модулей нас и будет интересовать. Поэтому под "модулем" мы будем понимать не любую часть программы, а только такую, которую можно оттранслировать независимо от других частей и затем объединить с ними. Как реализуется этот вариант в ЯА, и будет рассмотрено в данной главе.

12.1. Работа в системе MASM

Вначале рассмотрим, какие действия надо предпринять в системе программирования MASM, чтобы можно было оттранслировать и запустить на счет многомодульную программу. Попутно объясним, как транслируется и выполняется одномодульная программа.

Отметим, что здесь рассматриваются только основные правила работы в системе MASM; детальное описание системы можно найти в [6].

Как хранить многомодульную программу

Пусть мы разделили программу на несколько модулей, которые затем будем транслировать независимо друг от друга. Тогда каждый модуль необходимо записать в отдельный файл на диске. При этом файлам, в которых хранятся тексты программ на ЯА, принято давать расширение ASM; делать это не обязательно, но желательно. Мы будем считать, что модули программы размещены в файлах M1.ASM, M2.ASM и т. д.

Трансляция модулей

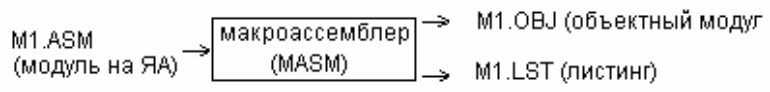
Как известно, все свои действия на ПК пользователь описывает в виде приказов операционной системе (ОС), которая постоянно находится в оперативной памяти и управляет работой ПК. В частности, с помощью таких приказов пользователь осуществляет трансляцию и запуск на счет своих программ.

Для того чтобы оттранслировать один модуль программы на ЯА, скажем модуль M1, надо дать следующий приказ операционной системе:

```
MASM M1.ASM, M1.OBJ, M1.LST;
```

(запятые и точка с запятой обязательны).

По этому приказу ОС считывает с диска макроассемблер и передает ему управление. Макроассемблер транслирует модуль из файла, указанного в качестве первого параметра приказа (из M1.ASM), и записывает оттранслированный вариант этого модуля, т. е. объектный модуль, в файл, указанный в качестве второго параметра приказа (в M1.OBJ), а в третий из указанных файлов (в M1.LST) записывается листинг программы:



В листинге указывается текст модуля как на ЯА, так и на машинном языке. Кроме того, если при трансляции модуля были обнаружены ошибки, то все они указываются в листинге. Поэтому, чтобы узнать об ошибках, надо будет затем посмотреть листинг. Собственно ради этого и нужен листинг, в дальнейшем он не используется.

Отметим, что имена и расширения файлов с объектным модулем и листингом могут быть любыми. Однако рекомендуется этим файлам давать то же имя, что было у файла с исходным модулем (M1), иначе при наличии многих модулей легко запутаться в том, какой файл к какому модулю относится. Это общий принцип именования файлов в ПК: файлам, относящимся к одной и той же программе (модулю), рекомендуется давать одинаковые имена, а вот различать эти файлы следует по расширениям. При этом обычно придерживаются определенной системы в использовании расширений, например, файлам с текстами программ на ЯА дают расширение ASM, файлам с объектными модулями – расширение OBJ (object), файлам с листингом – расширение LST (listing) и т. д. Выгода от этих расширений в том, что их можно опускать: если в качестве первого параметра приказа MASM указано имя файла без точки и расширения, то подразумевается именно расширение ASM, во втором параметре по умолчанию подразумевается расширение OBJ и т. д.

Далее. Если при трансляции модуля обнаружены ошибки (об этом выдается сообщение на экран ПК), то объектный модуль не создается, однако листинг будет создан. Следует просмотреть листинг, определить и исправить

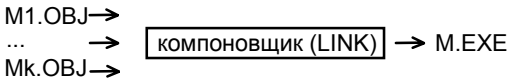
ошибки в исходном тексте модуля, а затем перетранслировать данный модуль с помощью того же приказа MASM.

Объединение (компоновка) модулей

Когда будут оттранслированы (без ошибок) все модули программы, их надо объединить в единую машинную программу. Это делается с помощью следующего приказа операционной системе:

```
LINK M1.OBJ+...+Mk.OBJ, M.EXE;
```

В первом параметре через плюс перечисляются названия всех файлов с объектными модулями программы, а как второй параметр указывается файл, в который компоновщик должен записать программу, полученную объединением указанных объектных модулей:



Отметим, что если файлы с объектными модулями имеют расширение OBJ, то это расширение вместе с точкой можно опустить. Файл же, в который записывается единая машинная программа, обязан иметь расширение EXE, а его имя может быть любым (обычно ему дают имя одного из модулей).

Запуск программы на счет

Компоновщик только объединяет модули в единую машинную программу и записывает в файл на диск, но не выполняет ее. Для запуска ее на счет надо задать приказ, состоящий только из названия того файла, где находится машинная программа:

```
M.EXE
```

(расширение EXE вместе с точкой можно и опустить).

По этому приказу ОС отыскивает на диске файл с указанным именем, записывает его в оперативную память и, считая, что в файле хранится машинная программа, передает ей управление. (Более точно, по приказу вызывается специальная программа – загрузчик, который и выполняет указанные действия, корректируя при этом вызванную машинную программу – настраивая ее на то место памяти, в которое она была помещена.) Получив управление, машинная программа начинает выполняться. Когда ее выполнение закончится (по команде FINISH), то управление возвращается в ОС и мы снова получаем право выполнить любой приказ ОС.

Отметим, что если программа уже оттранслирована (есть файл M.EXE), то ее можно выполнять любое число раз без повторной трансляции – достаточно набирать только приказ M.EXE.

Исправление ошибок

Если при выполнении программы обнаружена ошибка, тогда следует определить, в каком из модулей программы находится ошибка, и исправить этот модуль. Затем надо перетранслировать только этот модуль (по приказу MASM), не транслируя остальные модули. Далее надо снова обратиться к компоновщику (по приказу LINK), указав в нем все объектные модули программы. И, наконец, надо запустить программу на счет.

Как видно, здесь не надо перетранслировать модули, в которых нет ошибок. На этом экономится много времени, в этом и заключается выгода от независимой трансляции модулей.

Трансляция и выполнение одномодульной программы

Мы рассмотрели, как транслируется и выполняется программа, состоящая из нескольких модулей. А теперь рассмотрим, что делать с программой, состоящей только из одного модуля.

Ясно, что для такой программы не нужен этап объединения модулей в единую программу, что по исходному файлу можно сразу строить EXE-файл. Однако, чтобы не иметь двух схем трансляции (одну – для многомодульных программ, а другую – для одномодульных), принято компоновщик использовать всегда, даже если программа состоит только из одного модуля.

В связи с этим трансляция и счет одномодульной программы осуществляются с помощью следующих приказов (считаем, что текст программы на ЯА находится в файле M.ASM):

```
MASM M.ASM,M.OBJ,M.LST; – трансляция
LINK M.OBJ,M.EXE;       – "объединение"
M.EXE                   – счет
```

12.2. Модули. Внешние и общие имена.

Мы рассмотрели, что надо сделать для того, чтобы была оттранслирована и выполнена многомодульная программа. Теперь мы рассмотрим, как следует записывать сами модули, какие проблемы возникают в многомодульных программах и как они решаются.

12.2.1. Структура модулей. Локализация имен

Каждый модуль записывается так, как если бы он был самостоятельной программой, т. е. представляет собой последовательность предложений, оканчивающуюся директивой END:

```
<предложение>
...
<предложение>
END [<точка входа>]
```

Среди всех модулей программы выделяется один, называемый головным (главным). Это модуль, который должен выполняться первым, с него начинается выполнение программы, а уж он затем передает управление всем другим мо-

дулям. Синтаксически головной модуль отличается от других только тем, что в его директиве END указывается точка входа – метка той команды этого модуля, с которой должно начинаться выполнение программы. В остальных модулях точка входа не указывается. (Если точки входа все же указаны в нескольких модулях программы, то учитывается только та, что первой "попалась на глаза" компоновщику.)

Отметим очень важную особенность модулей: все имена, описанные в модуле, локализуются в нем. Это значит, что в разных модулях программы можно пользоваться одинаковыми именами и никаких конфликтов из-за этого не будет. Такая локализация очень выгодна, если модули составляются разными людьми, которые не должны бояться, что используемые ими имена совпадут.

12.2.2. Внешние и общие имена. Директивы EXTRN и PUBLIC

Хотя обычно в виде модулей описываются достаточно независимые друг от друга части программы, эти модули, конечно, как-то взаимодействуют друг с другом – делают переходы, обмениваются данными. И вот при таком взаимодействии возникает ряд проблем. Рассмотрим их.

Пусть в программе имеется два модуля M1 и M2. Пусть в M2 описана процедура P, к которой будет обращаться модуль M1, и пусть модуль M2, в свою очередь, пользуется переменной X и константой K из модуля M1:

```
; модуль M1      ; модуль M2
  ...              ...
X DB ?           P PROC FAR
K EQU 100          ...
  ...              MOV X,0
  CALL P           MOV AX,K
  ...              ...
```

При независимой трансляции модулей ассемблер сталкивается со следующей проблемой. Имена X и K принадлежат модулю M1, поэтому они описываются в модуле M1 и не должны описываться в модуле M2. Когда ассемблер транслирует модуль M2, то в это время он ничего не знает о модуле M1, об его именах. Поэтому, встретив в M2 имена X и K и не найдя их описания в M2, ассемблер, конечно, зафиксирует ошибку "неописанное имя". Чтобы не было такой ошибки, надо как-то сообщить ассемблеру, что имена X и K, хотя и используются в M2, описаны все же в другом модуле, откуда они и берутся. Такое сообщение делается с помощью директивы EXTRN (external, внешний), имеющей следующий вид:

```
EXTRN <имя>:<тип>, ..., <имя>:<тип>
```

где <тип> – это BYTE, WORD, DWORD, ABS, NEAR или FAR. (ABS – это стандартная константа ЯА со значением 0.) Данная директива может указываться любое число раз и в любых местах модуля.

В нашем конкретном случае в модуле M2 надо указать директиву

```
EXTRN X:BYTE, K:ABS
```

Она сообщает ассемблеру, что имена X и K являются, как говорят, внешними по отношению к данному модулю, т. е. берутся извне, из другого модуля, и потому ассемблер не должен искать описание этих имен в модуле M2.

Кроме того, в директиве указываются и типы имен. Например, в нашей директиве сказано, что X – имя переменной размером в байт. Без этой информации ассемблер не сможет правильно оттранслировать команды и директивы, содержащие имя X. В самом деле, как ассемблер должен транслировать команду MOV X,0 – как пересылку байта или как пересылку слова? Чтобы не было такой неопределенности, и надо указывать ассемблеру типы внешних имен. Типы BYTE, WORD и DWORD указываются для имен переменных, тип ABS – для имен констант, а типы NEAR и FAR – для меток или имен процедур. Например, директива

```
EXTRN P:FAR
```

которую, как следует из всего сказанного, надо поместить в модуль M1, сообщает, что P – это метка или имя процедуры из какого-то иного модуля, причем вызов ее должен быть дальним.

Отметим, что ответственность за согласованность типа внешнего имени, указанного в директиве EXTRN, и типа, которое имя получает при описании в "своем" модуле, несет автор программы. Например, если в модуле M1 имя K описано как имя константы, то и в модуле M2 внешнее имя K должно быть объявлено именем константы (указано с типом ABS).

Итак, поместив директиву EXTRN в модуль M2, мы сообщили ассемблеру, что имена X и K являются внешними, т. е. описанными в другом модуле (в каком именно – указывать не надо). Однако этого мало. Надо также в том модуле, где эти имена описаны (т. е. в M1), поместить директиву PUBLIC, перечислив в ней эти имена:

```
PUBLIC <имя>, ..., <имя>
```

Эта директива может указываться любое число раз и в любых местах модуля.

В нашем конкретном случае в модуле M1 надо записать директиву

```
PUBLIC X,K
```

Этой директивой мы сообщаем, что имена X и K данного модуля разрешено использовать в других модулях, что мы экспортируем эти имена. Имена, которые описаны в модуле и которые доступны другим модулям, по отношению к этому модулю называются общими (доступными всем, публичными).

Отметим, что имена, перечисленные в директиве PUBLIC, обязательно должны быть описаны в модуле и что в директиве их типы указывать не надо.

Может возникнуть вопрос: а зачем в модуле M1 указывать, что имена X и K являются общими? Разве недостаточно директивы EXTRN в модуле M2?

Оказывается, недостаточно. Дело здесь в следующем. Модули M1 и M2 транслируются по отдельности и объединяются лишь после того, как будут переведены на машинный язык. Но в машинном варианте модуля M1 не останется никаких имен, поэтому, когда при объединении модулей программы в модуле M2 надо будет заменять имена X и K на соответствующие адрес и число, сделать это уже не удастся – в модуле M1 нужной информации не будет. Так вот, чтобы при трансляции модуля M1 ассемблер сохранил нужную информацию об именах X и K, мы и указываем директивой PUBLIC на особую роль этих имен. Встретив эту директиву, ассемблер сохранит в объектном модуле M1 необходимую информацию об именах X и K, которой затем, при объединении модулей, и воспользуется компоновщик.

Итак, если в каком-то модуле программы некоторое имя описано как внешнее, то в каком-то другом модуле это имя обязательно должно быть объявлено общим. При этом никакое имя не должно быть объявлено общим в нескольких модулях программы, иначе будет путаница.

С учетом всего сказанного наши модули M1 и M2 должны иметь следующий вид:

```
; модуль M1          ; модуль M2
  EXTRN P:FAR          EXTRN X:BYTE,K:ABS
  PUBLIC X,K           PUBLIC P
  ...                  ...
X DB ?               P PROC FAR
K EQU 100              ...
  ...                  MOV X,0
  CALL P               MOV AX,K
  ...                  ...
```

12.2.3. Сегментирование внешних имен

Следующая проблема, которая возникает в многомодульных программах, – это проблема сегментирования внешних имен.

Обратите внимание, что в директиве EXTRN не указывается, по какому сегментному регистру надо сегментировать внешнее имя. А это важно знать ассемблеру, т. к. от этого зависит, как он должен транслировать команду, содержащую такое имя. Например, если имя X надо сегментировать по регистру ES, тогда команда MOV X,0 должна восприниматься как MOV ES:X,0, т. е. ассемблер должен добавлять префикс ES:. Но если имя X должно сегментироваться по регистру DS, тогда префикс DS:, который подразумевается в команде MOV по умолчанию, ассемблер не должен подставлять. Конечно, если перед внешним именем явно указан префикс (например, MOV ES:X,0), тогда проблем нет, но обычно имена указываются без префиксов, поэтому и возникает проблема с выбором по умолчанию сегментных регистров для внешних имен. В ЯА эта проблема решается следующим образом:

1) внешние имена констант (т. е. с типом ABS) не сегментируются;
2) внешние метки и имена процедур (т. е. с типом NEAR или FAR) всегда сегментируются по регистру CS, причем для дальних меток и процедур

(FAR) всегда формируются дальние переходы, а для близких (NEAR) – близкие переходы. Например:

```
EXTRN L:FAR,M:NEAR
...
CALL L  ; = CALL FAR PTR L
CALL M  ; = CALL NEAR PTR M
```

3) для внешних имен переменных (BYTE, WORD, DWORD) действуют следующие правила:
   - если директива EXTRN с именем размещена вне какого-либо программного сегмента, то любая команда с этим внешним именем транслируются без префикса, т. е. считается, что это имя должно сегментироваться по регистру, который в данной команде подразумевается по умолчанию;
   - если директива EXTRN с именем размещена внутри программного сегмента, тогда это внешнее имя по умолчанию сегментируется по тому же регистру, что и все имена из этого сегмента.

   Пример:

```
  EXTRN X:WORD
A SEGMENT
  EXTRN Y:WORD
  ...
A ENDS
B SEGMENT
  EXTRN Z:WORD
  ...
B ENDS
  ...
  ASSUME ES:A,DS:B
  ...
  INC X    ; = INC DS:X
  INC X[BP]  ; = INC SS:X[BP]
  INC Y    ; = INC ES:Y
  INC Y[BP]  ; = INC ES:Y[BP]
  INC Z    ; = INC DS:Z
  INC Z[BP]  ; = INC DS:Z[BP]
```

Таким образом, место для директивы EXTRN, вообще говоря, нельзя выбирать произвольно, от этого места зависит, как будут сегментироваться перечисленные в ней внешние имена.

Что же касается директивы PUBLIC, то ее можно размещать где угодно. Обычно ее указывают в самом начале модуля – так легче увидеть, какие имена из этого модуля экспортируются, являются общими.

12.2.4. Доступ к внешним именам

Указать, по какому сегментному регистру какое внешнее имя сегментируется, – это только полдела. Это позволяет правильно оттранслировать команды, содержащие внешние имена, но еще не обеспечивает правильного выполнения этих команд. Надо еще, чтобы при счете программы в сегментных регистрах находились нужные значения.

Рассмотрим такой пример:

```
; модуль M1          ; модуль M2
A SEGMENT              PUBLIC X
  EXTRN X:WORD       B SEGMENT
Y DW 0               X DW ?
A ENDS               B ENDS
  ...                  ...
  ASSUME DS:A
  ...
```

Предположим, что мы находимся в модуле M1, что регистр DS уже установлен на начало сегмента A и что мы хотим выполнить присваивание X:=Y. Тогда, казалось бы, надо выполнить команды

```
MOV AX,Y
MOV X,AX
```

Однако это не так. Почему? Первая из этих команд проработает правильно: имя Y воспринимается как сокращение адресной пары DS:Y, а эта пара и дает правильный адрес ячейки Y из сегмента A. Но вот следующая команда будет работать неправильно. Имя X воспринимается как сокращение пары DS:X, и это правильно, но вот текущее значение регистра DS не обеспечивает правильного доступа к ячейке X из сегмента B, ибо для правильного доступа к X регистр DS должен указывать не на начало сегмента A, а на начало сегмента B. Следовательно, прежде чем выполнить вторую команду, в регистр DS надо заслать начало сегмента B: DS:=B.

Как это сделать? Имя сегмента B не описано в модуле M1 и не является для него внешним, т. к. не указано в директиве EXTRN (отметим, что имена сегментов и не могут быть объявлены как внешние имена), поэтому указывать имя B в модуле M1 вообще нельзя. Но указывать явно это имя и не надо, если вспомнить про оператор SEG: SEG X – это как раз и есть начало того сегмента, где описано имя X, т. е. SEG X = B. Поэтому нам надо на самом деле выполнить присваивание DS:=SEG X.

С учетом этого и того, что, скорее всего, надо сохранять текущее значение (A) регистра DS, пересылка X:=Y должна реализовываться так:

```
MOV AX,Y          ;AX:=Y
PUSH DS           ;спасти DS (=A)
MOV BX,SEG X
MOV DS,BX         ;DS:=B
MOV X,AX          ;X:=AX
POP DS            ;восстановить DS (=A)
```

Как видно, получилось громоздко и неуклюже, причем все это приходится повторять при каждом обращении к внешнему имени. Почему так произошло? А потому, что мы пытаемся работать с разными сегментами данных, используя только один сегментный регистр. Из-за этого нам приходится устанавливать этот регистр на начало то одного сегмента, то другого, приходится то спасать этот регистр, то восстанавливать. Ясно, что мы существенно

облегчим себе жизнь, если для каждого сегмента будем использовать свой собственный регистр, например, регистр DS для сегмента A и регистр ES для сегмента B. Давайте так и сделаем: установим один раз регистр ES на начало внешнего сегмента B, а затем будем использовать ES при каждом обращении к именам из этого сегмента:

```
MOV AX,SEG X
MOV ES,AX
...
MOV AX,Y
MOV ES:X,AX
```

Отметим, что в этом случае уже нет необходимости специально помещать директиву EXTRN X:WORD внутрь сегмента A, никакой выгоды это нам не даст. Эту директиву можно разместить и вне программных сегментов, например, в самом начале модуля M1, где ее проще всего заметить.

Так обычно решается проблема доступа к внешним переменным. Что же касается доступа к внешним меткам или процедурам, то здесь ситуация существенно проще.

Рассмотрим такой пример:

```
; модуль M1          ; модуль M2
  EXTRN L:FAR           PUBLIC L
  ...                 C SEGMENT
  JMP L                   ...
  ...                 L: ...
```

Как уже было сказано, внешние метки всегда сегментируются по регистру CS, причем если они объявлены как дальние, то для них формируются дальние переходы. Поэтому в нашем примере команда JMP L будет восприниматься как команда JMP FAR PTR L, машинный эквивалент которой выглядит так:

```
КОП seg(L):ofs(L)
```

По этой машинной команде в регистр CS заносится seg(L)=C, а в регистр IP записывается ofs(L), т. е. переход в сегмент C на метку L произойдет правильно без каких-либо дополнительных мер с нашей стороны.

12.2.5. Пример многомодульной программы

Для иллюстрации всего сказанного рассмотрим конкретный пример многомодульной программы, которая вводит текст, содержащий не более 100 символов и оканчивающийся точкой, и затем выводит его в обратном порядке, заменив в нем все большие латинские буквы на соответствующие малые буквы.

Обычно в виде модуля описывают набор процедур и данных, используемых другими модулями. Так же поступим и мы, для чего разделим нашу программу на два модуля – головной и вспомогательный. Во вспомогательном модуле опишем переменную EOT, значением которой является символ, служащий концом ввода текста, и процедуру LOWLAT, заменяющую большую латинскую букву на малую. Головной же модуль будет вводить текст, записывать его в некоторый массив в обратном порядке, попутно заменяя с помощью функции LOWLAT большие буквы на малые, и в конце распечатает этот массив.

```
;вспомогательный модуль
       PUBLIC EOT, LOWLAT
    D1 SEGMENT
   EOT DB '.'           ;символ конца ввода
    D1 ENDS
    C1 SEGMENT
       ASSUME CS:C1
LOWLAT PROC FAR
;процедура перевода больших латинских букв в малые
;на входе: AL – любой символ
;на выходе: AL – малая буква, если AL – большая латинская
;                            (иначе AL не меняется)
       CMP AL,'A'  ;AL<'A' или AL>'Z' -> NOLAT
       JB NOLAT
       CMP AL,'Z'
       JA NOLAT
       ADD AL,-'A'+'a'  ;AL:=(AL-'A')+'a'
NOLAT: RET
LOWLAT ENDP
    C1 ENDS
       END
;головной модуль
       INCLUDE IO.ASM
       EXTRN EOT:BYTE, LOWLAT:FAR
     S SEGMENT STACK
       DB 128 DUP(?)
     S ENDS
     D SEGMENT
   TXT DB 100 DUP(?),'$'
     D ENDS
     C SEGMENT
       ASSUME SS:S, DS:D, CS:C
START: MOV AX,D
       MOV DS,AX        ;DS=D (для доступа к TXT)
       MOV AX,SEG EOT
       MOV ES,AX        ;ES=D1 (для доступа к EOT)
       MOV SI,100
       OUTCH '>'        ;приглашение к вводу
  INP: INCH AL          ;ввод символа
       CMP AL,ES:EOT    ;конец ввода?
       JE PR
       CALL LOWLAT      ;замена символа
       DEC SI
       MOV TXT[SI],AL   ;запись символа в конец TXT
       JMP INP
   PR: LEA DX,TXT[SI]
       OUTSTR           ;вывод TXT
       FINISH
     C ENDS
       END START
```

12.3. Параметры директивы SEGMENT

В многомодульной программе возникает еще одна проблема, связанная с взаимным расположением сегментов ее модулей. Если не предпринимать каких-либо специальных мер, то в объединенной программе сегменты будут располагаться в следующем порядке: сначала будут идти сегменты модуля, указанного первым в приказе LINK, по которому вызван компоновщик, затем будут идти сегменты модуля, указанного вторым, и т. д. Это не всегда удобно. Иногда нужно расположить сегменты в ином порядке или желательно объединение ("слияние") некоторых сегментов из разных модулей в единый сегмент. Так вот, в ЯА имеется средство, с помощью которого можно влиять на расположение сегментов в окончательной программе. Этим средством являются параметры директивы SEGMENT, с которой начинаются описания программных сегментов.

В общем случае директива SEGMENT имеет следующий вид:

```
<имя сегмента> SEGMENT [<выравнивание>] [<объединение>] ['<класс>']
```

Любые параметры могут отсутствовать, но если они есть, то должны быть перечислены в указанном порядке. Причем друг от друга эти параметры отделяются пробелами, а не запятыми.

Рассмотрим каждый из этих параметров.

12.3.1. Параметр "класс"

В этом параметре в одиночных кавычках указывается некоторое имя, которое рассматривается как имя класса, к которому мы решили отнести наш сегмент. Зачем нужны эти классы?

Как уже сказано, при объединении модулей в единую программу компоновщик будет располагать сегменты этих модулей в следующем порядке: сначала в программе будут идти все сегменты модуля, указанного в директиве LINK первым, затем – все сегменты второго модуля и т. д. Но иногда желательно размещать сегменты в ином порядке. Например, часто хотят разместить рядом "родственные" сегменты (скажем, сегменты команд или сегменты данных) из разных модулей. Так вот, если какие-то сегменты отнесены к одному классу, то в окончательной программе компоновщик расположит их рядом. Отметим при этом, что все сегменты, для которых не указан класс, считаются относящимися к одному и тому же классу с "пустым" именем.

Пусть, к примеру, в программе имеются модули M1 и M2, состоящие из следующих сегментов (см. слева; для краткости для каждого сегмента указывается только его директива SEGMENT):

Модуль M1	*Модуль M2*	*Программа M*	*Класс*
A SEGMENT	C SEGMENT 'DATA'	A	DATA
'DATA'	...	C	DATA
...	D SEGMENT 'CODE'	B	
B SEGMENT	...	E	
...	E SEGMENT	D	CODE
	...		

и пусть мы объединяем эти модули по приказу:

```
LINK M1.OBJ+M2.OBJ,M.EXE;
```

Тогда в программе M сегменты будут расположены так, как указано справа.

Почему именно так? Сначала компоновщик просматривает первый из указанных ему модулей, т. е. M1. Его сегменты, относящиеся к разным классам, не переупорядочиваются, а располагаются в том порядке, как они записаны в модуле: A B. Далее компоновщик просматривает сегменты модуля M2. Первый из них – сегмент C – имеет класс DATA. Компоновщик смотрит, был ли ранее сегмент того же класса. Да, был, это сегмент A, поэтому компоновщик размещает C вслед за A, "отодвигая" B: A C B. Следующий сегмент D имеет класс CODE, но среди предыдущих сегментов не было сегмента такого класса, поэтому компоновщик размещает D в конце последовательности: A C B D. Последний сегмент E относится к классу с "пустым" именем, а так как до этого уже был сегмент B того же класса, то компоновщик располагает его вслед за B и перед D. Поскольку других сегментов и модулей нет, то данное расположение сегментов является окончательным.

12.3.2. Параметр "объединение"

Этот параметр может принимать следующие значения: PUBLIC, STACK, AT и COMMON.

Значение PUBLIC

Как уже сказано, сегменты одного класса всегда располагаются в памяти рядом. Однако каждый из них сохраняет свою независимость. Это означает, что прежде чем обратиться к любому из этих сегментов, надо установить соответствующий сегментный регистр на начало данного сегмента. А это лишние команды, лишний расход времени. Этого можно избежать, если объединить сегменты в единый сегмент. Например, можно считать, что в нашей программе имелся только один сегмент данных, но по каким-то причинам мы описали его по частям в разных модулях, и вот теперь, при компоновке программы из модулей, мы собрали эти части вместе и получили единый исходный сегмент данных. Теперь достаточно один раз установить соответствующий сегментный регистр на начало единого сегмента и далее, не меняя значение регистра, осуществлять по нему доступ ко всем ячейкам сегмента.

Такое объединение сегментов происходит, если в качестве параметра "объединение" в директивах SEGMENT, начинающих эти сегменты, указано слово PUBLIC. Более точно правило объединения следующее: компоновщик объеди-

няет в один сегмент такие сегменты (из любых модулей), у которых одно и то же имя, которые относятся к одному и тому же классу и в директивах SEGMENT которых указан тип объединения PUBLIC. Если хотя бы одно из этих трех условий нарушено (у сегментов разные имена, разные классы или разные типы объединения), тогда сегменты не объединяются. В частности, сегмент, для которого не указан параметр "объединение", не будет объединяться ни с каким другим сегментом.

Пример:

Модуль M1	*Модуль M2*	*Модуль M3*	*Программа*
`A SEGMENT PUBLIC`	`C SEGMENT PUBLIC`	`A SEGMENT 'Q'`	`A(M1)`
`'Q'`	`'Q'`	`...`	`A(M2)`
`...`	`...`		
`B SEGMENT`	`A SEGMENT PUBLIC`		`C`
`...`	`'Q'`		`A(M3)`
	`...`		`B`

В окончательной программе сегменты A из модулей M1 и M2 объединены в один сегмент, а сегменты C из M2 и A из M3 располагаются рядом с этим общим сегментом, т. к. имеют тот же класс Q, но не объединяются с ним, поскольку у C иное имя, а у A из M3 не указан параметр PUBLIC. Отметим, что если бы сегмент A из модуля M2 не объединялся с сегментом A из модуля M1, то он располагался бы вслед за сегментом C, но объединяемые сегменты, конечно, размещаются рядом, "отодвигая" остальные сегменты.

Теперь уточним, в чем разница между объединением сегментов и просто расположением сегментов рядом в памяти. Пусть имеются два модуля:

```
;модуль M1              ; ofs      ;модуль M2            ; ofs
  EXTRN Z:WORD          ;            PUBLIC Z            ;
A SEGMENT PUBLIC 'Q'    ;          A SEGMENT PUBLIC 'Q'  ;
X DW ?                  ; 0        Z DW ?                ; 0
Y DB ?                  ; 2        A ENDS                ;
A ENDS                  ;            ...
  ...
```

и пусть регистр ES установлен на начало сегмента A из M1. Рассмотрим, как будет транслироваться и выполняться команда MOV BX,ES:Z из модуля M1.

Если бы оба сегмента A не были объединены (например, у одного из них не было бы параметра PUBLIC), тогда смещения имен в каждом из них отсчитывались бы от начала сегмента (см. колонки ofs). Поэтому имя Z в нашей команде было бы заменено на смещение 0:

```
MOV BX,ES:Z ==> MOV BX,ES:0
```

Поскольку, согласно нашему предположению, регистр ES указывает на сег-мент A из модуля M1, то эта команда проработает неправильно: в регистр BX будет записано значение переменной X, а не переменной Z. Для правильного доступа к переменной Z надо перед нашей командой установить регистр ES на начало сегмента A из модуля M2, а это лишние команды. И здесь не играет никакой роли, рядом ли в памяти расположены сегменты или нет.

Если же оба сегмента A объединены, тогда сегмент A из модуля M2 считается продолжением сегмента A из модуля M1:

```
A SEGMENT  ; ofs
X DW ?     ; 0
Y DB ?     ; 2
           ; (пропуск 13 байт)
Z DW ?     ; 10h
A ENDS
```

Это означает, что теперь смещения всех имен этого объединенного сегмента будут отсчитываться от начала данного сегмента и именно на эти смещения будут заменяться имена (таким пересчетом и коррекцией команд занимается компоновщик). Например, имя Z теперь будет заменяться не на смещение 0, а на смещение 10h. Почему на 10h (=16), а не на 3? Здесь надо вспомнить, что сегменты располагаются в памяти с адресов, кратных 16, и это правило действует, даже если сегменты объединяются. Поэтому сегмент A из M2 подсоединяется к сегменту A из M1, но не впритык, а с ближайшего адреса, кратного 16, и тем самым между переменными Y и Z будет оставлено свободными 13 байт (от этого зазора можно и избавиться – см. разд. 12.3.3). Итак, смещение имени Z равно 10h, поэтому в нашей команде имя Z будет заменено на 10h:

```
MOV BX,ES:Z ==> MOV BX,ES:10h
```

Эта команда уже проработает правильно, менять перед ней значение регистра ES не надо.

Таким образом, объединение (слияние) нескольких сегментов означает, что эти сегменты будут расположены в памяти рядом и, главное, смещения всех имен из этих сегментов будут отсчитываться от начала объединенного сегмента. Это и позволяет один раз установить какой-то сегментный регистр на начало общего сегмента и далее, не меняя значение регистра, использовать его для доступа ко всем именам общего сегмента.

Значение STACK

Если какие-то сегменты (из любых модулей) имеют одно и то же имя, относятся к одному и тому же классу и для каждого из них указан тип объединения STACK, то эти сегменты объединяются в один сегмент – так же, как и сегменты типа PUBLIC. (Отметим, что друг с другом сегменты типов PUBLIC и STACK не объединяются.) Разница между этими двумя типами проявляется в том, что объединенный сегмент типа STACK рассматривается как сегмент стека, и именно на него перед выполнением программы будут установлены регистры SS и SP (на сегмент типа PUBLIC никакие регистры автоматически не устанавливаются).

Если в программе нет ни одного сегмента типа STACK, то компоновщик выдаст на экран ПК предупреждение об этом. Если же имеется несколько таких сегментов, то регистры SS и SP устанавливаются на тот из них, который (или

часть которого, если он получен слиянием) последним "попался на глаза" компоновщику.

Значение AT <константное выражение>

В выражении не должно быть ссылок вперед. Значение выражения трактуется как номер некоторого сегмента памяти, т. е. как абсолютный адрес сегмента без последних четырех битов. Программный сегмент с параметром AT ни с кем не объединяется, а располагается в памяти по указанному адресу. Например, сегмент

```
VIDEO SEGMENT AT 0B800h
      DW 25*80 DUP(?)
VIDEO ENDS
```

будет расположен с абсолютного адреса 0B8000h и займет 2000 слов (здесь находится видеопамять, содержимое которой отображается на экран ПК).

С помощью AT-сегментов обычно вводятся ассемблерные обозначения для фиксированных участков оперативной памяти (вектора прерываний, видеопамяти и т. п.). Чтобы не менять содержимое этих участков, в состав AT-сегментов не должны входить предложения, порождающие машинный код, т. е. запрещены любые команды, директивы DB, DW и DD с операндами, отличными от ?, и т. п. Если подобные предложения все-таки имеются, тогда компоновщик блокирует запись в память их машинного кода.

Значение COMMON

Все сегменты, которые имеют одно и то же имя, относятся к одному и тому же классу и в директиве SEGMENT которых указан параметр COMMON, компоновщик размещает в памяти с одного и того же адреса (с адреса, по которому был размещен первый из этих сегментов), накладывая их содержимое друг на друга. Результирующий сегмент имеет длину наибольшего из этих сегментов.

COMMON-сегменты используются тогда, когда одни и те же ячейки памяти желательно именовать по-разному в разных модулях программы. Например, в программе, в состав которой входят следующие модули:

```
;модуль M1              ;модуль M2
S SEGMENT COMMON        S SEGMENT COMMON
A DB 1                  X DB 3
B DD 0                  Y DB ?
S ENDS                  S ENDS
  ...                     ...
```

на оба сегмента S будет отведено 5 байт, причем первый из этих байтов – это одновременно и байт A (с точки зрения модуля M1), и байт X (с точки зрения модуля M2), второй байт – это начало двойного слова B в модуле M1 и переменная Y в модуле M2, а оставшиеся три байта – продолжение переменной B (для модуля M1). При этом, если в приказе LINK модуль M2 был указан после мо-дуля M1, в первом байте результирующего сегмента окажется чис-

ло 3, значение второго байта будет неопределенным, а остальные три байта будут нулевыми.

12.3.3. Параметр "выравнивание"

Как мы уже видели, при объединении сегментов типа PUBLIC или STACK может образоваться зазор между сегментами. Если для необъединяемых сегментов такой пропуск необходим (независимый сегмент должен начинаться с адреса, кратного 16, чтобы на его начало можно было установить сегментный регистр), то для объединенных сегментов этот зазор – лишь потеря памяти. Поэтому в ЯА предусмотрено средство, позволяющее уничтожать такие зазоры, а точнее – регулировать начало размещения сегментов в памяти. Этим средством является параметр "выравнивание" директивы SEGMENT: он указывает, с адреса какой кратности должен начинаться сегмент. Возможные значения данного параметра и соответствующие им начальные адреса сегментов таковы:

```
BYTE   – ближайший свободный адрес;
WORD   – ближайший адрес, кратный 2;
PARA   – ближайший адрес, кратный 16 (параграф);
PAGE   – ближайший адрес, кратный 256 (страница).
```

Например, в программе, содержащей следующие модули:

```
;модуль M1                ;модуль M2
A SEGMENT PUBLIC 'Q'      A SEGMENT BYTE PUBLIC 'Q'
X DW ?                    Z DW ?
Y DB ?                    A ENDS
A ENDS                      ...
  ...
```

будет образован единый сегмент A:

```
A SEGMENT
X DW ?
Y DB ?
Z DW ?
A ENDS
```

причем содержимое сегмента A из модуля M2 будет без пропуска подсоединено к предложениям сегмента A из модуля M1, поэтому смещение имени Z будет равно 3, а не 10h, как было в примере из разд. 12.3.2.

Отметим, что если в директиве SEGMENT параметр "выравнивание" опущен, то по умолчанию берется значение PARA. Именно по этой причине у нас до сих пор все сегменты начинались с адресов, кратных 16.

13. ВВОД-ВЫВОД. ПРЕРЫВАНИЯ

Цель данной главы – рассмотреть, как в ПК осуществляется ввод-вывод и как можно реализовать те операции ввода-вывода (INCH, OUTINT и т. п.), которыми мы пользовались в предыдущей части книги.

13.1. Команды ввода-вывода

Все устройства ЭВМ принято делить на внутренние и внешние. Внутренние устройства – это центральный процессор (ЦП) и оперативная память, а внешние – все остальные устройства (внешняя память, клавиатура, дисплей, принтер и т. д.). В широком смысле, под вводом-выводом понимается обмен информацией между ЦП и любым внешним устройством. При этом "отсчет" ведется от ЦП: ввод – это передача данных в ЦП из внешнего устройства, а вывод – передача данных из ЦП во внешнее устройство.

В ПК передача информации между ЦП и внешними устройствами, как правило, осуществляется через порты. Порт – это некоторый регистр размером в байт или слово, причем этот регистр находится вне ЦП и не имеет никакого отношения к обычным регистрам типа AX, SI и т. п. Вообще-то все порты являются байтовыми, но два соседних порта рассматриваются как порт размером в слово. Порты нумеруются, их номера – от 0 до 0FFFFh. Таким образом, потенциально может быть 65536 портов, однако реально их значительно меньше (несколько десятков). С каждым внешним устройством связан свой порт или несколько портов, их номера заранее известны.

Как ЦП, так и внешнее устройство может записывать информацию в порт и может считывать из него. Со стороны ЦП (а точнее, той программы, которую он выполняет) эти операции осуществляются с помощью следующих машинных команд:

```
Чтение из порта (ввод): IN AL,n или IN AX,n
Запись в порт (вывод): OUT n,AL или OUT n,AX
```

По команде IN в регистр AL (AX) переносится содержимое порта с номером n, а команда OUT реализует обратное действие: в порт с номером n записывается содержимое регистра AL (AX).

Номер порта (n) в этих командах может быть задан двояко – либо явным числом от 0 до 255, либо регистром DX, значение которого и трактуется как номер порта. Например:

```
IN AL,97h     ;AL := порт 97h
MOV DX,836
OUT DX,AX     ;порт 836 := AX
```

Первый вариант операнда n используется, когда номер порта небольшой и известен заранее, а второй вариант – когда номер порта может быть любым или когда он становится известным только во время счета программы.

Сценарий ввода-вывода через порты существенно зависит от специфики того внешнего устройства, с которым ведется обмен, но достаточно типичным является следующий сценарий.

Часто ЦП связан с внешним устройством двумя портами: через один передаются данные – это порт данных, а через другой пересылается всякого рода управляющая информация – это порт управления. Когда ЦП (а точнее, программа, которую он выполняет) хочет начать обмен с каким-то внешним устройством, то он записывает в порт управления этого устройства определенную комбинацию битов, которая трактуется как призыв к установлению связи. Если устройство функционирует и не занято, то в ответ оно записывает в этот порт другую комбинацию битов, сигнализирующую о том, что оно готово к обмену. ЦП же, выждав определенное время, считывает информацию из порта и, если там оказался сигнал о том, связь установлена, начинает собственно обмен.

Пусть ЦП хочет что-то вывести. Тогда он записывает в порт управления комбинацию символов, трактуемую как команда "выводи", а в порт данных – выводимую величину (например, код символа). Внешнее же устройство, считав из портов эту информацию, записывает в порт управления сигнал "занято" и начинает собственно вывод (например, печатает символ на бумаге). В это время ЦП либо переходит в состояние ожидания, опрашивая (считывая) в цикле порт управления, либо занимается другой работой – до тех пор, пока в порте управления не сменится сигнал "занято". Когда внешнее устройство закончит вывод, то оно записывает в этот порт сигнал об успешном завершении своей работы или сигнал об ошибке (например, порвалась бумага в принтере). Далее ЦП продолжает свою работу (например, осуществляет следующую операцию обмена) или как-то реагирует на сбой в работе устройства.

Осуществлять подобным образом ввод-вывод в каждой программе – занятие трудоемкое. Оно требует знания многих технических деталей – номеров портов, управляющих сигналов и сигналов ответа, порядка опроса портов и т. п., причем эта информация различна для разных внешних устройств. А кроме того, в каждой новой программе приходится заново описывать все действия, связанные с вводом-выводом, что, конечно, накладно. В то же время в большинстве программ используются в общем-то одни и те же операции ввода-вывода. Учитывая все это, поступают так: один раз описываются часто используемые операции ввода-вывода, которые скрывают всю "кухню" работы с портами, и эти операции включают в состав операционной системы (ОС), постоянно находящейся в ОП, чтобы ими могла пользоваться любая программа, выполняемая на ЭВМ. Такой способ существенно упрощает жизнь программистов, поэтому обычно используют только эти операции и не пользуются портами напрямик. С портами работают лишь тогда, когда надо реализовать какой-нибудь необычный, экзотический ввод-вывод.

В связи с этим мы больше не будем говорить о машинных командах ввода-вывода и портах, а рассмотрим некоторые из операций ввода-вывода, которые включены в состав ОС.

13.2. Прерывания. Функции DOS

13.2.1 Прерывания

Одна из основных задач ОС – управление работой всех устройств, входящих в состав ЭВМ. Для выполнения этой задачи ОС обязана, в частности, оперативно следить за событиями, происходящими в устройствах.

Пусть, к примеру, ЦП выполняет некоторую программу и пусть в это время в каком-то внешнем устройстве произошло событие (например, на клавиатуре нажата клавиша), на которое ОС должна прореагировать (например, запомнить код нажатой клавиши). Естественно, ждать пока закончится выполнение текущей программы нельзя, она может работать еще долго и за это время может быть нажато много других клавиш, так что информация о первой из нажатых клавиш будет потеряна. Надо сразу, оперативно прореагировать на это событие. Как это сделать?

В современных ЭВМ эта проблема решается с помощью прерываний. Когда в некотором устройстве происходит событие, достойное внимания ОС, это устройство посылает в ЦП специальный сигнал, называемый сигналом прерывания или просто прерыванием. Получив такой сигнал, ЦП прерывает свою обычную работу, т. е. выполнение команд текущей программы, и передает управление на ОС, которая определяет, какое событие произошло, и соответствующим образом реагирует на него. Например, если прерывание пришло от клавиатуры в связи с тем, что была нажата некоторая клавиша, то ОС, считав из порта клавиатуры код этой клавиши, запоминает его где-то в своем участке памяти. Когда ОС закончит обработку прерывания, она заставляет ЦП возобновить выполнение прерванной программы с той команды, которую ЦП ранее должен был выполнить, но из-за поступившего прерывания не выполнил. И вот так ОС "вступает в игру" всякий раз, когда в устройствах ЭВМ возникают те или иные события. Это и позволяет ОС следить за всеми событиями в ЭВМ и оперативно реагировать на них.

Уточним некоторые детали этой схемы на примере ПК.

Как запоминается то место текущей программы, в котором было прервано ее выполнение? Напомним, что в ПК адрес команды, которая должна быть выполнена следующей, находится в регистрах CS и IP. В момент прерывания эти регистры и показывают на ту команду, которая была бы выполнена, если бы не было прерывания, и с которой, следовательно, надо будет возобновить работу программы. (Отметим, что ЦП воспринимает сигнал прерывания только между командами, но не во время выполнения команды.) Поэтому, когда приходит сигнал прерывания, ЦП запоминает значения этих двух регистров, причем запоминает в стеке текущей программы: сначала в стек записывается содержимое CS, а затем – IP.

Замечание. Ранее уже отмечалось, что если машинная программа сама не использует стек, то все равно под него надо отводить место в памяти; причина этого, как видно, в том, что стек программы используется при прерываниях, которые происходят во время ее счета, например, из-за случайного нажатия клавиш на клавиатуре.

Помимо адреса следующей команды программы процессор также запоминает в стеке и текущее значение флага регистров Flags, т. е. значения всех флагов. Зачем? Дело в том, что программа может быть прервана между командой сравнения и следующей за ней командой условного перехода, и позже придется возобновлять работу программы с этой команды условного перехода. Ясно, что в этот момент все флаги должны быть такими же, какими они были после команды сравнения, т. е. в момент прерывания. Учитывая это, ЦП и спасает регистр флагов, чтобы затем можно было его восстановить. При этом регистр Flags записывается в стек до адреса прерывания.

Итак, когда в ЦП приходит сигнал прерывания, то он приостанавливает выполнение текущей программы и спасает в стеке этой программы текущее значение флага регистров и адрес следующей команды, а затем уже передает управление на ОС. (Отметим, что ЦП также обнуляет флаги прерывания IF и трассировки TF.)

Теперь уточним, как происходит передача управления на ОС. Отметим, что ОС – это обычная машинная программа, хотя и выполняющая некоторые специфические действия. Поэтому, чтобы заставить ее работать, надо передать управление на одну из ее команд. Но на какую именно? Ведь на разные прерывания надо реагировать по-разному, поэтому при разных прерываниях надо передавать управление на разные команды ОС. В ПК эта проблема решается следующим образом.

Все возможные прерывания нумеруются числами от 0 до 255. Для каждого прерывания составляется своя процедура обработки прерывания (ПОП), и все эти процедуры включаются в состав ОС. Начальные адреса этих процедур записываются в самые первые ячейки памяти; эти адреса попадают сюда при загрузке ОС в оперативную память. При этом каждый такой адрес записывается в виде пары seg:ofs, где seg – номер сегмента памяти, в котором находится соответствующая ПОП, а ofs – ее смещение внутри этого сегмента. В связи с этим каждый адрес занимает 4 байта, двойное слово, и потому начальный адрес i-й ПОП располагается в двойном слове с адресом 4*i. Всего эти начальные адреса занимают 4*256=1024=1 Кб. Это место памяти принято называть вектором прерываний. Таким образом, вектор прерываний – это массив из 256 элементов, i-й элемент которого – начальный адрес процедуры обработки прерывания с номером i.

Как используется вектор прерываний? Устройство, в котором произошло событие, достойное внимания ОС, посылает в ЦП не только сигнал прерывания, но и номер этого прерывания (устройства, конечно, знают номера своих прерываний). Если это номер i, тогда ЦП, сохранив в стеке регистр флагов и

адрес прерывания, просто напросто передает управление по адресу из i-го элемента вектора прерывания, делая тем самым переход на i-ю ПОП, которая и начинает заниматься обработкой данного прерывания.

ПОП – это в общем-то обычная процедура. В частности, если она должна использовать какие-то регистры, то, как и все "порядочные" процедуры, она в начале своей работы спасает в стеке (прерванной программы или своем внутреннем) значения этих регистров, а в конце работы восстанавливает их. Поэтому, когда будет возобновлена работа прерванной программы, в регистрах останутся те же значения, которые были в них в момент прерывания.

И, наконец, последнее уточнение. Когда ПОП закончит свою работу, она должна возобновить работу прерванной программы. Как это сделать? Очень просто: из стека надо считать три слова и восстановить по ним регистры IP и CS (загрузка в эти регистры адреса и означает переход по этому адресу) и регистр флагов. Для осуществления этих действий в систему команд ПК введена команда "возврат из прерывания" (interrupt return) без операндов:

```
IRET
```

Ее действие: стек -> IP, стек -> CS, стек -> Flags. Этой командой процедуры обработки прерываний и завершают свою работу.

Итак, что произошло? Прежние значения всех регистров сохранены, прежние состояния всех флагов восстановлены, управление передано на ту команду прерванной программы, которую ЦП из-за прерывания не успел выполнить ранее. А это значит, что данная программа продолжит свою работу так, как будто бы и не было прерывания.

13.2.2. Функции DOS

Рассмотренный механизм прерываний первоначально был предназначен для того, чтобы ОС могла следить за событиями во внешних устройствах ЭВМ (в принтере, дисководах и др.). Но затем поняли, что этот механизм можно использовать и в других целях.

Например, с его помощью можно реагировать на события, происходящие в самом ЦП. Пусть, к примеру, при выполнении какой-то программы встретилась команда деления на 0. Ясно, что это ошибка и на нее надо как-то реагировать. И делается это так: тот блок ЦП, который реализует операцию деления, обнаружив, что делитель равен 0, вырабатывает сигнал прерывания с определенным номером. Дальнейшие же события, несмотря на то, что сигнал прерывания послал сам себе ЦП, идут по рассмотренной схеме, а потому вызывается соответствующая ПОП из ОС, которая выдает на экран сообщение об ошибке "деление на 0", прекращает выполнение программы и передает управление на ту часть ОС, которая осуществляет прием и выполнение приказов пользователя.

Другой случай, где полезно использование прерываний, особенно интересен для нас сейчас. Давно уже замечено, что в различных программах приходится выполнять одни и те же действия, например, выводить символы на эк-

ран или вводить их с клавиатуры, причем реализация этих действий требует детального знания тех или иных устройств ЭВМ. Так вот, чтобы избавить авторов программ от знания этих деталей, избавить их от необходимости выписывать эти действия в каждой программе заново, такие часто повторяющиеся действия описывают один раз в виде соответствующих процедур и включают их в состав ОС, и теперь все программы могут пользоваться ими.

Но как обращаться к этим процедурам? Конечно, можно использовать обычную команду вызова процедур (CALL), но для этого надо знать начальные адреса этих процедур. А эти адреса, к сожалению, меняются от одной версии ОС к другой, причем версии ОС меняются достаточно часто, поэтому сегодня эти адреса одни, а завтра – другие. Учитывая это, процедуры ОС принято вызывать несколько иначе – с помощью прерываний, исходя из следующего соображения: раз ОС "вступает в игру" при любом прерывании, то почему бы не воспользоваться прерываниями и для вызова ее процедур.

Конкретно эта идея реализуется так. Определенные номера прерываний оставляют незанятыми для прерываний от устройств ЭВМ и предназначают для вызова процедур ОС, для чего в соответствующие элементы вектора прерываний записываются начальные адреса этих процедур. Поэтому, если выработать прерывание с одним из этих номеров, то будет вызвана соответствующая процедура. При этом не надо знать точный адрес этой процедуры, а достаточно знать только номер прерывания, по которой она вызывается. Номера же эти остаются одними и теми же во всех версиях ОС.

Но для вызова таким образом процедур ОС надо уметь генерировать прерывания из обычных программ. Для этого в систему команд ПК включена специальная команда – команда прерывания (interrupt):

```
INT i8
```

Эта команда вызывает искусственное, насильственное прерывание с номером i8 (0<=i8<=255), а это значит, что начнет работать та процедура ОС, начальный адрес которой записан в элементе вектора прерываний с номером i8. Закончив свою работу, эта процедура вернет управление на команду программы, следующую за командой INT. Таким образом, эта команда очень похожа на команду CALL, но только вызывает процедуры более "хитрым" способом.

Отметим, что по команде INT можно сгенерировать прерывание с любым номером, например, прерывание, соответствующее делению на 0 или нажатию клавиши на клавиатуре. Однако делать этого не следует, а надо использовать эту команду только для вызова процедур ОС.

Далее. В состав ОС (имеется в виду дисковая операционная система фирмы Microsoft – MS DOS) входит много процедур и для них не хватает допустимых номеров прерываний. В связи с этим процедуры объединяются в группы с тем, чтобы процедуры из одной группы вызывались по прерыванию с одним и тем же номером. Процедуры одной группы принято называть функ-

циями соответствующего прерывания. Чтобы различать их, перед выполнением команды INT в регистр AH записывают номер нужной функции:

```
MOV AH,<номер функции>
INT <номер прерывания>
```

Получив по команде INT управление, ОС по номеру из регистра AH определяет, к какой именно функции произошло обращение, после чего и передает управление ей.

Для выполнения функции может потребоваться определенная информация (например, для функции вывода символа на экран нужно указать код этого символа). Такая информация передается через регистры. Какие именно параметры надо передавать и через какие регистры – зависит от конкретной функции, общих правил здесь нет.

Мы не будем перечислять номера всех прерываний и функций ОС, это можно найти в многочисленных книгах по MS DOS. Мы рассмотрим лишь некоторые функции, реализующие ввод-вывод и окончание счета программы, – те, на основе которых чуть позже будут определены операции ввода-вывода, которыми мы пользуемся в данной книге. Отметим, что все они являются функциями прерывания с шестнадцатеричным номером 21 (десятичным 33).

13.2.3. Некоторые функции прерывания 21h

Завершение программы

Завершив все свои действия, программа обязана вернуть управление операционной системе, чтобы пользователь мог продолжить работу на ПК. Такой возврат, соответствующий операции "завершить программу", реализуется функцией 4Ch прерывания 21h:

```
MOV AL,<код завершения>
MOV AH,4Ch
INT 21h
```

Каждая программа, вообще говоря, обязана сообщить, успешно или нет она завершила свою работу. Дело в том, что любая программа вызывается из какой-то другой программы (например, из операционной системы), и иногда вызвавшей программе, чтобы правильно продолжить работу, надо знать, выполнила ли вызванная программа все, что надо, или она проработала с ошибкой. Такая информация передается в виде кода завершения программы (некоторого целого числа), который должен быть нулевым, если программа проработала правильно, и ненулевым (каким именно – оговаривается в каждом случае особо) в противном случае. (Узнать код завершения вызванной программы можно с помощью функции 4Dh прерывания 21h.) Потребуется этот код или нет, программа все равно должна выдать его.

Вывод на экран (в текстовом режиме)

Для вывода одного символа на экран ПК используется функция 02 прерывания 21h:

```
MOV DL,<код выводимого символа>
MOV AH,2
INT 21h
```

Выводимый символ высвечивается в позиции курсора (что бы там ни было записано), после чего курсор сдвигается на одну позицию вправо. Если курсор находился в конце строки экрана, то он перемещается на начало следующей строки, а если курсор был в конце последней строки экрана, то содержимое экрана сдвигается на одну строку вверх, а внизу появляется пустая строка, в начало которой и устанавливается курсор.

Особым образом осуществляется вывод символов с кодами 7, 8, 9, 10 (0Ah) и 13 (0Dh). Символ с кодом 7 (bell, звонок) на экране не высвечивается (и курсор не сдвигается), а вызывает звуковой сигнал. Символ с кодом 8 (backspace, шаг назад) возвращает курсор на одну позицию влево, если только он не был в самой левой позиции строки. Символ с кодом 9 (tab, табуляция) смещает курсор вправо на ближайшую позицию, кратную 8. Символ с кодом 10 (line feed, перевод строки) перемещает курсор в следующую строку экрана, оставляя его в той же колонке. Символ с кодом 13 (carrige return, возврат каретки) устанавливает курсор на начало текущей строки; вывод подряд символов с кодами 13 и 10 означает перевод курсора на начало следующей строки.

Для вывода на экран строки (последовательности символов) можно, конечно, использовать функцию 02, однако сделать это можно и за один прием с помощью функции 09 прерывания 21h:

```
DS:DX := начальный адрес строки
MOV AH,9
INT 21h
```

Перед обращением к этой функции в регистр DS должен быть помещен номер того сегмента памяти, в котором находится выводимая строка, а в регистр DX – смещение строки внутри этого сегмента. При этом в конце строки должен находиться символ '$' (код 24h), который служит признаком конца строки и сам не выводится.

Среди функций DOS нет такой, которая выводит числа. Такую операцию, если надо, приходится реализовать на основе рассмотренных функций.

Ввод с клавиатуры

При нажатии (в любой момент) клавиши на клавиатуре ее код (соответствующий символ) заносится операционной системой в специальный буфер ввода, откуда эти коды будут затем считываться функциями ввода. Тем самым возможен "досрочный" ввод: символы могут быть набраны еще до того, как программа начнет ввод. Размер этого буфера ОС – 15 позиций, поэтому

если до выполнения какой-нибудь функции ввода уже было нажато 15 клавиш, то на нажатие 16-й и последующих клавиш ОС не реагирует (кроме выдачи звукового сигнала). По мере считывания из этого буфера коды сдвигаются в его начало и коды новых нажатых клавиш дописываются в конец буфера; функции ввода всегда считывают символы из начала буфера. Если вызвана функция ввода, а в буфере нет никаких символов, то ОС ожидает нажатия клавиши.

Если программа не желает вводить символы, которые были набраны досрочно, то она должна очистить буфер ввода ОС, для чего следует выполнить функцию 0Ch (12) прерывания 21h при нулевом регистре AL:

```
MOV AL,0
MOV AH,0Ch
INT 21h
```

Из всех функций DOS, реализующих ввод с клавиатуры, мы рассмотрим только одну – функцию 0Ah (10) прерывания 21h, с помощью которой можно ввести сразу несколько символов (строку) и которая допускает редактирование набираемого текста:

```
DS:DX := адрес буфера для записи введенной строки
MOV AH,0Ah
INT 21h
```

Эта функция вводит символы до тех пор, пока не будет нажата клавиша Enter, и записывает их в указанный буфер. Вводимые символы высвечиваются на экране (ввод с эхом). Пока не нажата клавиша Enter, набираемый текст можно редактировать с помощью следующих клавиш:

←, Backspace – отмена последнего символа
Esc – отмена всего набранного текста

Буфер, начальный адрес которого задается в регистрах DS и DX (это не буфер ОС, а буфер программы), должен иметь следующую структуру:

max	n	s1	s2	...	sn	CR		...	
0	1	2	3		n+1	n+2			max+1

Перед обращением к функции в начальный байт (с индексом 0) буфера должно быть записано число (max), указывающее максимальное количество символов, которое имеет право ввести эта функция (размер буфера должен быть рассчитан на это количество). Если уже введено max-1 символов, то следующие символы не вводятся и не записываются в буфер, а лишь раздается звуковой сигнал (max-ым символом будет код клавиши Enter).

Число (n) реально введенных символов записывается функцией в байт буфера с индексом 1, а сами символы – начиная с позиции, имеющей индекс 2. В (n+2)-ю позицию буфера записывается "конец строки" CR (код 13), соответствующий клавише Enter, однако в числе n он не учитывается.

Пример:

```
BUF DB 10, ?, 10 DUP(' ')    ;в сегменте данных
    ...
    LEA DX,BUF
    MOV AH,0Ah
    INT 21h
```

Если при вводе были набраны символы ABC, то содержимое буфера BUF будет следующим:

```
BUF[0]=10, BUF[1]=3, BUF[2]=41h (код A), BUF[3]=42h,
BUF[4]=43h,
BUF[5]=13 (CR), остальные байты не изменятся
```

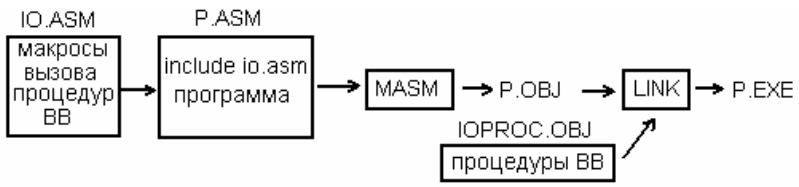
13.3. Операции ввода-вывода

Как видно, функции ОС реализуют очень полезные, но, к сожалению, слишком мелкие операции ввода-вывода. Например, с их помощью можно вывести на экран символ, но нельзя вывести число. Это не очень приятно. Во-первых, более крупные операции (типа вывода чисел) приходится реализовывать в программах каждый раз заново. Во-вторых, отсутствие таких операций вызывает серьезные трудности при изучении языка ассемблера, т. к. без этих операций ввода-вывода составлять даже простые, но законченные программы нельзя, а рассказывать о том, как реализуются эти операции, на начальном этапе изучения языка еще рано. В данной книге, чтобы решить эти проблемы, мы предположили, что кто-то для нас уже описал подходящие процедуры ввода-вывода, и активно ими пользовались во всех наших программах практически с самого начала изучения ЯА. Теперь пришла пора привести описание всех этих операций.

В данном разделе рассмотрен один из возможных вариантов того, как можно определить эти операции. Это как бы образец, по которому можно реализовать и иной пакет операций ввода-вывода.

13.3.1. Схема хранения и подключения операций ввода-вывода

Вначале рассмотрим схему хранения операций ввода-вывода (ВВ) и подключения их к программам:



Часть операций ввода-вывода реализована в виде процедур, описания которых собраны в отдельном модуле, хранящемся в файле с именем IOPROC.ASM (текст модуля приведен ниже). Предполагается, что этот мо-

дуль заранее оттранслирован и в виде объектного модуля записан в файл с именем IOPROC.OBJ.

Данный модуль должен быть подсоединен к программе на этапе ее компоновки. Для этого надо вызвать компоновщик по следующему приказу (считаем, что программа одномодульна и хранится в файле P):

```
LINK P.OBJ+IOPROC.OBJ, P.EXE;
```

Таким образом, полный цикл обработки программы (трансляция, компоновка и счет) реализуется следующими тремя приказами ОС:

```
MASM P.ASM,P.OBJ,P.LST;
LINK P.OBJ+IOPROC.OBJ, P.EXE;
P.EXE
```

Если бы были только процедуры из модуля IOPROC, тогда в программах пришлось бы выписывать по несколько команд обращения к этим процедурам. Например, обращение к процедуре вывода знакового числа выглядит так:

```
MOV AX,x          ;выводимое число
MOV DL,n          ;ширина поля вывода
CALL PROCOUTINT   ;вызов процедуры
```

Это достаточно громоздко и не очень удобно, т. к. нужно помнить о том, через какой регистр какой параметр надо передавать. Для того чтобы сделать вызовы этих процедур более короткими и наглядными, обращения ко всем этим процедурам оформлены как макросы. Например, указанные три команды описаны в виде макроса OUTINT, поэтому вместо них достаточно выписать только одну строчку – макрокоманду

```
OUTINT x,n
```

которая при трансляции программы автоматически заменится на эти команды.

Отметим, что некоторые операции (например, FINISH) столь просты, в них всего две-три команды, что их невыгодно реализовывать в виде процедур. Так вот, и эти операции описаны в виде макросов, только в этом случае макрос определяет не обращение к процедуре, а саму операцию. Описание всех этих макросов собрано в отдельном файле с названием IO.ASM (текст его также приведен ниже).

Поскольку в этом файле собраны только макроопределения, а по ним, как известно, не формируются никакие машинные команды, то данный файл не образует модуль, который можно было бы транслировать отдельно. Поэтому данный файл подключается к программе не на этапе компоновки, как модуль IOPROC.OBJ, а еще на этапе трансляции – по директиве INCLUDE IO.ASM. По ней в текст программы вставляются все макроопределения из файла, после чего в программе уже можно пользоваться макрокомандами типа OUTINT x,n и FINISH.

Далее приводятся тексты файлов IOPROC.ASM и IO.ASM без дополнительных пояснений, поскольку все используемые в них приемы программирования так или иначе были рассмотрены в предыдущих частях книги, а кроме того, эти тексты снабжены необходимыми комментариями. Отметим лишь, что директивы .XLIST и .LIST, используемые в файле IO.ASM, объясняются в разд. 14.4.

13.3.2. Текст файла IOPROC.ASM

```
; Модуль IOPROC: Процедуры ввода-вывода
public procnl, procoutnum, procflush, procinch, procinint
iocode segment
       assume cs:iocode
;*********************************************
;          ВЫВОД НА ЭКРАН
;*********************************************
;=======================================
; Перевод курсора на новую строку экрана
; Обращение: call procnl
; Параметров нет
;---------------------------------------
procnl proc far
       push dx
       push ax
       mov ah,2
       mov dl,13        ; CR (курсор на начало строки)
       int 21h
       mov dl,10        ; LF (курсор на следующую строку)
       int 21h
       pop ax
       pop dx
       ret
procnl endp
;=============================================================
==
; Вывод целого числа-слова со знаком или без знака
; Обращение: call procoutnum
; На входе: ax - выводимое число
;           dh - число со знаком (1) или без знака (0)
;           dl - ширина поля вывода (>=0)
;         (если поле больше, чем надо, то слева добавляются
;          пробелы, если меньше - выводится только число)
;-------------------------------------------------------------
--
procoutnum proc far
       push bp
       mov bp,sp
       push ax
       push dx
       push si
```

```
       sub sp,6          ; отвести 6 байт в стеке под число
; учет знака
       cmp dh,1          ; вывод со знаком (dh=1)?
       jne pon0
       cmp ax,0
       jge pon0
       mov dh,2          ; если вывод со знаком и ax<0,
       neg ax            ; то dh:=2, ax:=abs(ax)
 pon0: push dx           ; спасти dh (знак) и dl (ширину)
; запись цифр числа в стек (в обратном порядке)
       xor si,si         ; si - кол-во цифр в числе
 pon1: mov dx,0          ; ax -> (dx,ax)
       div cs:ten        ; ax=ax div 10; dx=ax mod 10
       add dl,'0'
       mov [bp-8+si],dl ; цифра -> стек
       inc si
       or ax,ax
       jnz pon1          ; еще не 0
; запись минуса, если есть, в стек
       pop dx
       cmp dh,2
       jne pon2
       mov byte ptr [bp-8+si],'-'
       inc si
; печать пробелов впереди
 pon2: mov dh,0          ; dx - ширина поля вывода
       mov ah,2          ; функция 02 прерывания 21h
pon21: cmp dx,si
       jle pon3          ; ширина <= длина числа
       push dx
       mov dl,' '
       int 21h
       pop dx
       dec dx
       jmp pon21
; печать (минуса и) цифр
 pon3: dec si
       mov dl,[bp-8+si]
       int 21h
       or si,si
       jnz pon3
; выход из процедуры
       add sp,6
       pop si
       pop dx
       pop ax
       pop bp
       ret
   ten dw 10
procoutnum endp
```

```
;*********************************************************
;                  ВВОД С КЛАВИАТУРЫ
;*********************************************************
; буфер ввода с клавиатуры (для работы с функцией 0Ah)
  maxb db 128         ; макс. размер буфера ввода
 sizeb db 0           ; число введенных символов в буфере
   buf db 128 dup(?) ; сам буфер ввода
  posb db 0           ; номер послед. считан. символа из buf
;=====================================================
; вспомогательная процедура ввода строки символов
; (включая Enter) в буфер buf (ввод без приглашения)
;-----------------------------------------------------
readbuf proc near
       push ax
       push dx
       push ds
       mov dx,cs
       mov ds,dx
       lea dx,buf-2  ; ds:dx – адрес buf[-2]
       mov ah,0Ah    ; ввод строки в буфер (включая Enter)
       int 21h
       call procnl   ; курсор на новую строку экрана
       inc cs:sizeb  ; в длине учесть Enter
       mov cs:posb,0 ; сколько символов уже считано из buf
       pop ds
       pop dx
       pop ax
       ret
readbuf endp
;====================================
; Очистка буфера ввода с клавиатуры
; Обращение: call procflush
; Параметров нет
;------------------------------------
procflush proc far
       push ax
       mov cs:sizeb,0   ; очистка buf
       mov cs:posb,0
       mov ah,0Ch       ; очистка DOS-буфера
       mov al,0
       int 21h
       pop ax
       ret
procflush endp
;=============================================================
; Ввод символа (с пропуском или без пропуска Enter)
; Обращение: call procinch
; На входе: al– Enter пропустить (0) или выдать как символ (1)
; На выходе: al – введенный символ (ah не меняется)
;-------------------------------------------------------------
```

```
procinch proc far
       push bx
princh1:
       mov bl,cs:posb       ; номер последнего считанного сим-
вола
       inc bl               ; след. номер
       cmp bl,cs:sizeb      ; не последний символ буфера?
       jb princh2
       jne princh10         ; буфер не считан до конца?
       cmp al,0             ; считывать ли конец строки
(Enter)?
       jne princh2
princh10:
       call readbuf         ; доввод в буфер
       jmp princh1          ; повторить
princh2:
       mov cs:posb,bl       ;запомнить номер считываемого сим-
вола
       mov bh,0
       mov al,cs:buf[bx-1]  ;al:=символ
       pop bx
       ret
procinch endp
;=============================================================
; Ввод целого числа (со знаком и без) размером в слово
; Обращение: call procinint
; На входе: нет
; На выходе: ax – введенное число
;-------------------------------------------------------------
procinint proc far
       push bx
       push cx
       push dx
; пропуск пробелов и концов строк вначале
prinint1:
       mov al,0
       call procinch ; al – очередной символ
                     ; (с пропуском Enter)
       cmp al,' '    ; пробел?
       je prinint1
; проверка на знак
       mov dx,0      ; dx – вводимое число
       mov cx,0      ; ch=0 – нет цифры, cl=0 – плюс
       cmp al,'+'
       je prinint2
       cmp al,'-'
       jne prinint3
       mov cl,1      ; cl=1 – минус
; цикл по цифрам
prinint2:
       mov al,1
       call procinch ; al – очередной символ (Enter – символ)
```

```
prinint3:            ; проверка на цифру
       cmp al,'9'
       ja prinint4   ; >'9' ?
       sub al,'0'
       jb prinint4   ; <'0' ?
       mov ch,1      ; ch=1 - есть цифра
       mov ah,0
       mov bx,ax     ; bx - цифра как число
       mov ax,dx     ; ax - предыдущее число
       mul cs:prten  ; *10
       jc provfl     ; >FFFFh (dx<>0) -> переполнение
       add ax,bx     ; +цифра
       jc provfl
       mov dx,ax     ; спасти число в dx
       jmp prinint2  ; к след. символу
; кончились цифры (число в dx)
prinint4:
       mov ax,dx
       cmp ch,1      ; были цифры?
       jne prnodig
       cmp cl,1      ; был минус?
       jne prinint5
       cmp ax,8000h  ; модуль отриц. числа > 8000h ?
       ja provfl
       neg ax        ; взять с минусом
prinint5:
       pop dx        ; выход
       pop cx
       pop bx
       ret
 prten dw 10
;---------- реакция на ошибки при вводе числа
provfl: lea cx,prmsgovfl    ; переполнение
       jmp prerr
prnodig: lea cx,prmsgnodig  ; нет цифр
prerr: push cs              ; печать сообщения об ошибке
       pop ds               ; ds=cs
       lea dx,prmsg
       mov ah,9             ; outstr
       int 21h
       mov dx,cx
       mov ah,9             ; outstr
       int 21h
       call procnl
       mov ah,4Ch           ; finish
       int 21h
 prmsg db 'Ошибка при вводе числа: ','$'
prmsgovfl db 'переполнение','$'
prmsgnodig db 'нет цифры','$'
procinint endp
iocode ends
       end                  ; конец модуля ioproc
```

13.3.3. Текст файла IO.ASM

```
.xlist     ;запрет записи этого файла в листинг
; Файл с макросами ввода-вывода, подключаемый
; к программе по директиве: include io.asm
;*********************************************************
;                 ОКОНЧАНИЕ СЧЕТА ПРОГРАММЫ
;*********************************************************
;=======================================================
; Окончание счета программы
; обращение: finish
; на входе: al – код завершения (можно игнорировать)
;-------------------------------------------------------
finish macro
       mov ah,4Ch
       int 21h
       endm
;*********************************************************
;             ВЫВОД НА ЭКРАН (в текстовом режиме)
;*********************************************************
;==========================
; Переход на новую строку
;       обращение: newline
;--------------------------
extrn procnl:far
newline macro
        call procnl
        endm
;==============================
; Вывод символа
;       обращение: outch c
;       где c – i8, r8 или m8
;------------------------------
 outch macro c
       push dx
       push ax
       mov dl,c
       mov ah,2
       int 21h
       pop ax
       pop dx
       endm
;=========================================================
; Вывод строки символов
;       обращение: outstr
;       на входе: ds:dx – начальный адрес строки
;       (в конце строки д.б. символ $, код 36 (24h))
;---------------------------------------------------------
outstr macro
       push ax
```

```
       mov ah,9
       int 21h
       pop ax
       endm
;=============================================================
; Вывод целого со знаком размером в слово
;       обращение: outint num [,leng]
;       где num – выводимое число: i16, r16, m16
;       leng – ширина поля вывода: i8, r8, m8 (со значением
>=0)
; Особенности вывода:
; если поле больше, чем надо, то слева добавляются пробелы,
; если меньше – выводится только число (целиком);
; по умолчанию leng=0
;-------------------------------------------------------------
extrn procoutnum:far
outint macro num,leng
       outnum <num>,<leng>,1
       endm
;=========================================
; Вывод целого без знака размером в слово
; обращение: outword num [,leng]
;            num и leng – как в outint
;-----------------------------------------
outword macro num,leng
        outnum <num>,<leng>,0
        endm
;-------------------------------------------------
; Вспомогательный макрос проверки написания имени
; разными (большими и малыми) буквами
;-------------------------------------------------
  same macro name,variants,ans
       ans=0
       irp v,<variants>
       ifidn <name>,<v>
       ans=1
       exitm
       endif
       endm
       endm
;-------------------------------------------------------------
; Вспомогательный макрос для outint (sign=1) и outword (=0)
;-------------------------------------------------------------
outnum macro num,leng,sign
       local regdx?
       push ax
       push dx
       same <num>,<dx,DX,Dx,dX>,regdx?
       if regdx?        ;;out dx,leng -->
       ifb <leng>       ;; mov al,leng
       mov al,0         ;; xchg ax,dx
```

```
       else
       mov al,leng
       endif
       xchg ax,dx
       else              ;;out num,leng (num<>dx) -->
       ifb <leng>        ;; mov dl,leng
       mov dl,0          ;; mov ax,num
       else
       mov dl,leng
       endif
       mov ax,num
       endif
       mov dh,sign
       call procoutnum  ;;ax=num, dl=leng, dh=sign
       pop dx
       pop ax
       endm
;************************************************************
;                    ВВОД С КЛАВИАТУРЫ
;************************************************************
;====================================
; Очистка буфера ввода с клавиатуры
;          обращение: flush
;------------------------------------
extrn procflush:far
 flush macro
       call procflush
       endm
;============================================================
; Ввод символа (с пропуском концов строк, т. е. Enter)
; обращение: inch x
;          где x - r8, m8
; на выходе: x - введенный символ
;------------------------------------------------------------
extrn procinch:far
  inch macro x
       local regax?
       same <x>,<ah,AH,Ah,aH>,regax?
       if regax?
       xchg ah,al     ;;x=ah
       mov al,0
       call procinch
       xchg ah,al
       else
       same <x>,<al,AL,Al,aL>,regax?
       if regax?
       mov al,0       ;;x=al
       call procinch
       else
       push ax        ;;x - не ah и не al
       mov al,0
       call procinch
```

```
       mov x,al
       pop ax
       endif
       endif
       endm
;=============================================================
; Ввод целого числа (со знаком и без) размером в слово
; обращение: inint x
;      где x - r16, m16
; на выходе: x - введенное число
; особенности ввода:
;   пропускаются все пробелы и концы строк перед числом;
;   число должно начинаться с цифры, перед ней возможен знак;
;   при минусе число вводится как отрицательное;
;   ввод идет до первой нецифры (в т.ч. до Enter),
;                                          она глотается;
;   при ошибке будет печать сообщения и останов программы;
; ошибки: "нет цифры" - в числе нет ни одной цифры
;         "переполнение" - большое по модулю число
;         (вне отрезка [-32768,+65535])
;-------------------------------------------------------------
extrn procinint:far
 inint macro x
       local regax?
       same <x>,<ax,AX,Ax,aX>,regax?
       if regax?
       call procinint            ;;x=ax
       else
       push ax                   ;;x<>ax
       call procinint
       mov x,ax
       pop ax
       endif
       endm
;=============================================================

; восстановить запись в листинг:
       .list
```

14. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

В этой главе рассматриваются те команды ПК и средства ЯА, которые относительно редко используются на практике и которые не были рассмотрены в предыдущих главах книги.

14.1. Двоично-десятичные числа

В гл. 1 уже отмечалось, что в ПК помимо целых чисел, представленных в двоичной системе счисления (двоичных чисел), используются и двоично-десятичные числа (binary coded decimal, BCD). В этом разделе рассматриваются команды ПК, предназначенные для реализации арифметических операций над такими числами.

14.1.1. Представление двоично-десятичных и ASCII-чисел

Напомним, что машинное представление двоично-десятичных чисел строится по следующему правилу: каждая цифра из десятичной записи (неотрицательного) целого числа заменяется на четверку битов, изображающих эту цифру в двоичной системе счисления. Соответствие между десятичными цифрами и четверками битов следующее:

10	*2*	*10*	*2*
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

На каждое двоично-десятичное число отводят соседние байты памяти – столько, сколько надо. Порядок, в котором цифры числа занимают эти байты, вообще говоря, не фиксируется и определяется программистом. Дело в том, что обработка этих чисел ведется по цифрам и переход от цифры к цифре организует сам программист, а он может с одинаковым успехом переходить от очередного байта как к следующему байту, так и к предыдущему. Но для определенности мы в дальнейшем будем располагать левые (старшие) цифры числа в байтах с меньшими адресами, а правые цифры – с большими.

В ПК используются два формата представления двоично-десятичных чисел – упакованный и неупакованный. При упакованном формате в каждый байт записываются две соседние цифры числа (при нечетном количестве цифр слева к числу приписывается 0), а при неупакованном формате в каждом байте размещается только одна цифра, которая прижимается к правому краю байта. Ниже приведены представления числа 592 в упакованном (слева) и неупакованном форматах (A – адрес первого из байтов, занятых числом):

В ЯА переменные, значения которых трактуются как двоично-десятичные числа, можно описать по директиве DB с несколькими операндами:

```
  PACK DB 5h, 92h     ;число 592 в упакованном виде
UNPACK DB 5, 9, 2     ;число 592 в неупакованном виде
```

Обратите внимание, что для упакованного числа в одном операнде надо указывать две цифры, причем указывать их в шестнадцатеричной системе, чтобы каждая из цифр при трансляции заменялась на соответствующую четверку битов независимо от другой цифры (если в качестве операнда указать десятичное число 92, тогда оно заменится на 0101100b, а не на 10010010b).

В реальных программах двоично-десятичные числа редко выписываются явно. Они обычно получаются из так называемых ASCII-чисел – из числовых строк, из записи чисел в виде символьных строк (например, '592'), и в конце снова преобразуются в такие строки. Эти преобразования реализуются очень просто: поскольку в системе кодировки ASCII символы-цифры от '0' до '9' имеют коды от 30h до 39h, то для преобразования цифры-символа в цифру-число надо лишь выделить 4 правых бита из кода этого символа, а для обратного преобразования цифру-число надо сложить (арифметически или логически) с 30h.

14.1.2. Сложение двоично-десятичных чисел

Теперь рассмотрим, как реализуются арифметические операции над двоично-десятичными числами. Начнем со сложения.

Сложение неупакованных двоично-десятичных чисел

В ПК нет команды сложения двоично-десятичных чисел как единых объектов, поэтому такие числа приходится складывать по цифрам: сначала складывают последние (младшие) цифры, затем – предпоследние и т. д. При этом, учитывая специфику представления таких чисел, при сложении двух их цифр приходится особым образом устанавливать, есть ли перенос в старшие цифры и какую цифру надо записать в соответствующую позицию числа-результата. Здесь возможны два случая. Если сумма двух цифр не превосходит 9 (например, 2+3=5=00000101b), тогда переноса в старшие цифры нет (в флаг переноса CF надо записать 0), а в качестве цифры результата надо взять получившуюся величину. Но если сумма цифр больше 9 (5+8=13=00001101b), тогда перенос в старшие цифры есть и его надо запомнить в флаге CF, а в результат надо записать величину, которая на 10 меньше полученной суммы. В последнем случае можно поступить и иначе: надо к сумме цифр прибавить 6 и затем обнулить левую половину байта (13+6=19=00010011b --> 00000011b=3); отметим, что ПК действует именно по этому варианту.

Такой анализ и преобразование суммы двух цифр двоично-десятичных чисел реализует следующая команда ПК:

```
ASCII-коррекция после сложения (ASCII adjust after addition): AAA
```

Эта команда ставится после команды сложения (например, ADD), которая сложила две десятичные цифры и записала сумму в регистр AL. Этот регистр является неявным операндом команды AAA, именно его содержимое она анализирует и корректирует. Например:

```
MOV AL,5
MOV AL,8   ;AL=13
AAA        ;CF=1, AL=03
```

С помощью данной команды можно следующим образом сложить два N-значных неупакованных двоично-десятичных числа X и Y и записать сумму в Z (команда CLC обнуляет флаг переноса – см. разд. 14.2):

```
   N EQU ...          ;N>0
   X DB N DUP(?)
   Y DB N DUP(?)
   Z DB ?,N DUP(?) ;первый байт – для переноса из старших цифр
    ...
;сложение неупакованных 2-10-х чисел: Z:=X+Y
     MOV CX,N         ;количество цифр в слагаемых
     MOV SI,N-1       ;индекс цифр слагаемых (справа налево)
     CLC          ;CF – перенос из младших цифр (вначале CF=0)
SUM: MOV AL,X[SI]
     ADC AL,Y[SI]     ;сложение цифр с учетом переноса
                      ;из млад. цифр
     AAA              ;коррекция суммы и запись в CF переноса
     MOV Z[SI+1],AL   ;запись цифры суммы в Z
     DEC SI           ;к предыдущим цифрам слагаемых
     LOOP SUM         ;(команды DEC и LOOP не меняют CF)
     MOV Z,0
     ADC Z,0          ;запись переноса из старших цифр
```

Теперь приведем полное описание действия команды AAA: если флаг дополнительного переноса AF равен 1 или если в 4 правых битах регистра AL находится величина от 10 до 15, тогда значение AL увеличивается на 6, флаг переноса CF получает значение 1 и значение регистра AH увеличивается на 1, иначе лишь обнуляется флаг CF; в любом случае 4 левых бита AL обнуляются. Отметим, что команда AAA меняет и другие флаги (например, флаг AF получает то же самое значение, что и флаг CF), однако это обычно не представляет интерес.

Поясним действия команды AAA. Прежде всего отметим, что команда анализирует только 4 правых бита регистра AL, не обращая внимания на левые биты (причина этого будет указана ниже). Поэтому, если предыдущая команда сложения получила сумму больше 9, то команда AAA сможет распознать только суммы от 10 до 15. Для распознавания же сумм больше 15 привлекается флаг дополнительного переноса AF. Этот флаг меняется мно-

гими командами, однако интерес представляет только его изменение в командах сложения и вычитания: в командах сложения (ADD, ADC, INC) флаг AF получает значение 1, если при сложении 4 младших битов операндов получилась сумма больше 15 (если есть перенос из этих битов в более старшие биты), а в командах вычитания (SUB, SBB, DEC) флаг AF равен 1, если величина 4 младших битов первого операнда меньше величины таких же битов второго операнда (если был заем единицы для младших битов первого операнда). Так вот, по флагу AF команда AAA и распознает случай, когда сумма, полученная в предыдущей команде, больше 15. Итак, распознав, что сумма больше 9, команда AAA корректирует ее – добавляет к ней 6 и обнуляет левую половину регистра AL, что, как уже отмечалось, соответствует получению "правильной" цифры результата. Отметим, что левая половина AL обнуляется и в случае, когда сумма не превосходит 9, поэтому в регистре AL всегда остается только "правильная" цифра. Что же касается увеличения на 1 значения регистра AH, то это можно рассматривать как еще один способ фиксирования переноса (если вначале в AH был 0), но вообще-то такое изменение AH нужно для реализации умножения двоично-десятичных чисел (см. разд. 14.1.4).

Сложение ASCII-чисел

Несколько усложненное действие команды AAA объясняется тем, что она ориентирована на сложение цифр не только неупакованных двоично-десятичных чисел, но и ASCII-чисел. Например, если по команде ADD сложить коды цифр 5 и 8, т. е. коды 35h и 38h, то получим сумму 6Dh = 01101101b. Если после этого выполнить команду AAA, то она будет обращать внимание только на правую половину суммы, т. е. на величину 1101b (=13), которая и является настоящей суммой цифр (как чисел). Поскольку эта величина больше 9, то команда AAA фиксирует в флаге CF перенос и прибавляет 6 к AL (6Dh+6=73h), после чего обнуляет левую половину этой суммы, получая "правильную" цифру 3. Осталось только объединить ее с числом 30h, чтобы получить ASCII-код этой цифры – 33h.

Если значения описанных в последнем примере переменных X, Y и Z рассматривать как ASCII-числа, тогда сложение Z:=X+Y можно реализовать следующим образом (замечание: поскольку команда OR портит флаг CF, то перенос в старшие цифры будем производить через регистр AH):

```
;сложение ASCII-чисел (числовых строк): Z:=X+Y
     MOV CX,N   ;количество цифр в слагаемых
     MOV SI,N-1 ;индекс цифр слагаемых (справа налево)
     MOV AH,0   ;AH – перенос из младших цифр (вначале его нет)
SUM: MOV AL,X[SI]
     ADD AL,Y[SI]
     ADD AL,AH  ;сложение цифр с учетом переноса из млад. цифр
     MOV AH,0
     AAA        ;коррекция сложения и запись переноса в AH
     OR AL,30h  ;преобразование цифры в символ
```

```
    MOV Z[SI+1],AL  ;запись кода цифры в Z
    DEC SI
    LOOP SUM
    OR AH,30h
    MOV Z,AH        ;запись в Z переноса из старших цифр
```

Сложение упакованных двоично-десятичных чисел

И здесь приходится складывать числа по частям, но уже не по одной цифре, а по паре цифр, упакованных в одном байте: по команде ADD или ADC складывается пара цифр из первого слагаемого с парой цифр из второго слагаемого, а затем с помощью специальной команды корректируется полученная сумма. Это следующая команда:

```
Десятичная коррекция после сложения
          (decimal adjust after addition): DAA
```

Эта команда имеет смысл, только если она поставлена после команды сложения, которая сложила упакованную пару правильных десятичных цифр и записала сумму в регистр AL. Команда DAA преобразует значение этого регистра в две правильные десятичные цифры и фиксирует в флаге CF перенос из этих цифр. Например:

```
MOV AL,59h
ADD AL,69h    ;AL=0C2h
DAA           ;CF=1, AL=28h
```

Более точно команда DAA действует так: если AF=1 или если величина в 4 правых битах AL больше 9, тогда AL увеличивается на 6 (это коррекция правой цифры пары); если CF=1 или если величина в 4 левых битах AL больше 9, то к AL прибавляется число 60h (т. е. 6 к левой половине) и CF получает значение 1, а иначе CF получает значение 0 (это коррекция левой цифры и учет переноса из данной пары цифр). Команда меняет и другие флаги, но их значения обычно не представляют интерес.

Например, в приведенных выше командах сумма правых цифр (9 и 9) равна 18 = 12h, поэтому в правую половину регистра AL попадает цифра 2, а цифра 1 как перенос добавляется к сумме левых цифр (5+6+1=12=0Ch), при этом флаг AF получает значение 1. По этому значению AF команда DAA и узнает, что правая цифра в AL неправильная, и увеличивает ее на 6; в этот момент AL=0C8h. Далее анализируется левая половина регистра AL: поскольку в команде ADD не было переноса, то CF=0, но в левых битах AL находится величина 0Ch, которая больше 9, поэтому левую цифру также надо скорректировать, для чего к ней прибавляется 6 (с записью переноса в CF): (12+6) mod 16 = 2 и CF=1.

Для примера приведем алгоритм сложения описанных выше чисел X, Y и Z, когда их значения трактуются как 2*N-значные упакованные числа:

```
;сложение упакованных 2-10-ных чисел: Z:=X+Y
    MOV CX,N    ;количество байтов (пар цифр) в слагаемых
    MOV SI,N-1  ;индекс байта слагаемых (справа налево)
```

```
     CLC         ;CF – перенос из младших цифр (вначале его нет)
SUM: MOV AL,X[SI]
     ADC AL,Y[SI];сложение двух пар цифр с учетом переноса
     DAA         ;коррекция цифр суммы и запись в CF переноса
     MOV Z[SI+1],AL  ;запись пары цифры суммы в Z
     DEC SI          ;(команды DEC и LOOP
     LOOP SUM        ; не меняют флаг CF)
     MOV Z,0
     ADC Z,0         ;запись в Z переноса из старших цифр
```

14.1.3. Вычитание двоично-десятичных чисел

Здесь все аналогично сложению. Вычитание двоично-десятичных чисел также осуществляется по цифрам, от младших цифр к старшим. Для коррекции же разности цифр в ПК используются команды AAS и DAS.

```
ASCII-коррекция после вычитания (ASCII adjust after
subtraction): AAS
```

Эту команду следует применять после команды вычитания (SUB и т. п.), которая записывает разность двух десятичных цифр в регистр AL. Команда AAS действует так: если AF=1 или если величина 4 правых битов AL больше 9, тогда из AL вычитается 6, из AH вычитается 1 и флаг CF получает значение 1 (фиксируется заем из старших цифр), иначе же флаг CF получает значение 0; в любом случае обнуляются 4 левых бита регистра AL. Например:

```
;коррекция не нужна
MOV AL,05h
SUB AL,02h      ;AL=03h, AF=0
AAS             ;AL=03h, CF=0
;коррекция нужна
MOV AL,02h
SUB AL,05h      ;AL=0FDh, AF=1
AAS             ;0FDh-6=0F7h --> AL=07h, CF=1
```

Команда AAS применяется при вычитании как неупакованных двоично-десятичных чисел, так и ASCII-чисел. Алгоритмы вычитания аналогичны алгоритмам сложения, поэтому мы их не будем приводить.

При вычитании упакованных двоично-десятичных чисел для коррекции цифр разности используется команда

```
Десятичная коррекция после вычитания
          (decimal adjust after subtraction): DAS
```

Эту команду следует применять после команды вычитания (SUB и т. п.), которая из одного байта, где упакованы две правильные десятичные цифры, вычитает другой подобный байт и записывает разность в регистр AL. Команда DAS действует так: если AF=1 или если величина 4 правых битов AL больше 9, тогда (для коррекции правой цифры разности) из AL вычитается 6; если CF=1 или если величина 4 левых битов AL больше 9, тогда (для коррекции левой цифры и учета заема) из AL вычитается 60h и флаг CF получает значение 1, а иначе CF получает значение 0. Например:

```
MOV AH,0
MOV AL,25h
SUB AL,52h ;AL=0D3h, AF=0, CF=1
DAS        ;AL:=0D3h-60h=73h, CF=1
```

14.1.4. Умножение и деление двоично-десятичных чисел

Надо сразу отметить, что реализовать в ПК умножение и деление двоично-десятичных чисел чрезвычайно сложно, т. к. система команд ПК предоставляет средства только для умножения и деления одной цифры на другую. Причем эти цифры обязательно должны быть неупакованными. По этой причине в ПК имеется только по одной команде коррекции для умножения и для деления.

Для коррекции при умножении используется команда

```
ASCII-коррекция после умножения (ASCII adjust after
multiply): AAM
```

Эту команду следует применять после команды умножения (MUL) двух неупакованных десятичных цифр, которая записывает их произведение в регистр AL. Команда AAM делит значение AL на 10 и записывает неполное частное в регистр AH, а остаток – в регистр AL, получая тем самым в двух этих регистрах правильные десятичные цифры произведения. Например:

```
MOV AL,6
MOV BH,9
MUL BH     ;AL=54=36h (AH=0)
AAM        ;AH=5, AL=4
```

В качестве примера использования команды AAM рассмотрим умножение N-значного неупакованного двоично-десятичного числа X на однозначное число Y с записью произведения в Z. Будем брать справа налево цифры из X и по очереди умножать их на Y, записывая в Z правую цифру получившегося произведения и прибавляя левую цифру к произведению старших цифр:

```
   N EQU ...          ;N>0
   X DB N DUP(?)
   Y DB ?             ;от 0 до 9
   Z DB ?, N DUP(?)   ;первый байт – для переноса
    ...
;умножение неупакованного 2-10-го числа X на десятичн. цифру Y
     MOV CX,N         ;количество цифр в X
     MOV SI,N-1       ;индекс цифры из X (справа налево)
     MOV BH,0         ;перенос из младших цифр
MLT: MOV AL,X[SI]
     MUL Y            ;умножение цифры из X на Y
     AAM              ;AH=левая цифра произведения, AL=правая
     ADD AL,BH        ;учет переноса из младших цифр
     AAA           ;коррекция сложения (если надо, AH:=AH+1)
     MOV Z[SI+1],AL   ;запись правой цифры в Z
     MOV BH,AH        ;запомнить перенос в старшие цифры
     DEC SI
     LOOP MLT
     MOV Z,BH         ;старшая цифра произведения
```

Для коррекции при делении двоично-десятичных чисел в ПК используется команда

```
ASCII-коррекция перед делением (ASCII adjust before
division): AAD
```

Эта команда ставится после команды, которая в регистре AX получает пару неупакованных цифр делимого (в AH – старшая цифра, в AL – младшая). Команда AAD преобразует эту пару цифр в двоичное число, записывая его в регистр AX: AX:=10*AH+AL. После этого уже можно делить обычным образом (по команде DIV) это число на делитель из одной десятичной цифры. Например:

```
MOV BH,9        ;делитель
MOV AX,0307h    ;делимое 37 (AH=3, AL=7)
AAD             ;AX=25h (37 как двоичное число)
DIV BH          ;AH=1 (37 mod 9), AL=4 (37 div 9)
```

В качестве примера рассмотрим деление N-значного неупакованного двоично-десятичного числа X на однозначное число Y с записью неполного частного от деления в Q, а остатка – в R.

Предлагается следующий алгоритм. Пусть надо разделить X=746 на Y=3. Берем 0 и первую цифру (7) числа X и делим это двузначное число (07) на Y; неполное частное (2) записываем в начало Q, а к остатку (1) присоединяем следующую цифру (4) числа X. Полученное двузначное число (14) делим на Y, после чего неполное частное (4) записываем в Q, а к остатку (2) присоединяем последнюю цифру (6) из X. Разделив двузначное число 26 на Y, неполное частное (8) записываем в Q, а остаток (2) – в переменную R.

```
  N EQU ...         ;N>0
  X DB N DUP(?)
  Y DB ?            ;от 0 до 9
  Q DB N DUP(?)     ;место для X div Y
  R DB ?            ;место для X mod Y
    ...
;деление неупакованного 2-10-го числа X на десятичную цифру Y
    MOV CX,N        ;количество цифр в X
    MOV SI,0        ;индекс цифры из X (слева направо)
    MOV AH,0        ;AH – остаток от деления старших цифр
DV: MOV AL,X[SI]    ;AL – очередная цифра из X
    AAD             ;AX=двоичное число из цифр AH и AL
    DIV Y           ;деление: AH=остаток, AL=неполное частное
    MOV Q[SI],AL    ;запись неполного частного в Z
    INC SI
    LOOP DV
    MOV R,AH        ;остаток от деления X на Y
```

14.2. Дополнительные команды ПК

Здесь вкратце перечислены команды ПК, которые не были рассмотрены в предыдущих главах. Это либо команды, которые достаточно редко исполь-

зуются на практике, либо команды, которые применяются в ситуациях, не рассматриваемых в данной книге (например, при взаимодействии центрального процессора с арифметическим сопроцессором).

```
Загрузка флагов в регистр AH (load AH from Flags): LAHF
Чтение флагов из регистра AH (save AH into Flags): SAHF
```

Команда LAHF записывает в регистр AH правую половину регистра флагов Flags, а команда SAHF записывает содержимое AH в правую половину регистра Flags.

Эти команды введены для совместимости с процессором 8080 (предшественником процессора 8086) и практически не используются.

```
Очистка флага переноса (clear carry flag): CLC
Установка флага переноса (set carry flag): STC
Изменение флага переноса (complement carry flag): CMC
```

По команде CLC флаг CF обнуляется (CF:=0), по команде STC флаг CF получает значение 1 (CF:=1), а по команде CMC значение флага CF меняется на противоположное (CF:=1-CF). Остальные флаги не меняются.

Эти команды могут использоваться перед командами циклического сдвига RCL и RCR для установки нужного значения флага CF перед сдвигом. Кроме того, поскольку с помощью этих команд значение флага CF можно менять достаточно просто, то флаг CF иногда используется для фиксации того, выполнено или нет некоторое условие; например, процедура может этим флагом сообщить, успешно она проработала или нет.

```
Очистка флага прерывания (clear interrupt flag): CLI
Установка флага прерывания (set interrupt flag): STI
```

По команде CLI флаг IF обнуляется (IF:=0), а по команде STI флаг получает значение 1 (IF:=1).

Установка флага IF в 1 означает, что процессору разрешено реагировать на поступающие к нему прерывания, а при значении 0 процессор не будет реагировать на прерывания (кроме прерываний из-за ошибок в работе аппаратуры). Эти команды обычно используются в процедурах операционной системы, занимающихся обработкой прерываний (при обработке одного прерывания приходится игнорировать другие прерывания).

```
Прерывание при переполнении (interrupt if overflow): INTO
```

Если перед этой командой флаг переполнения OF оказался равным 1, то она вырабатывает прерывание с номером 4 (работает как команда INT 4).

В ПК при выполнении сложения и вычитания знаковых чисел возможно переполнение мантиссы (ее абсолютная величина превышает допустимый размер и "залезает" на знаковый бит, изменяя его на противоположный), при котором получается неправильный результат. Однако процессор не фиксирует в этом случае ошибку, а лишь засылает 1 в флаг OF. "Поймать" такую

ошибку можно командой условного перехода, а можно и с помощью команды INTO. Она ставится сразу же за арифметической командой, например:

```
ADD AX,Y
INTO
```

и при переполнении мантиссы (при OF=1) вырабатывает прерывание с номером 4, которое можно обработать по-своему, если в 4-й элемент вектора прерываний (по адресу 0000:0010h) заранее записать начальный адрес процедуры обработки этого прерывания. (При начальной загрузке вектора прерываний в этот элемент записывается адрес некоторой ячейки, содержащей команду IRET, т. е. по умолчанию нет никакой реакции на это прерывание.)

```
"Пустая" команда (no operation): NOP
```

Команда NOP ничего не делает.

Эта однобайтовая команда обычно используется для того, чтобы "забить" команды, ставшие ненужными, не сдвигая при этом остальные команды программы, либо для того, чтобы сначала занять место в программе, в которое затем будут занесены "настоящие" команды.

```
Останов (halt): HLT
```

По этой команде процессор приостанавливает свою работу и ждет поступления любого прерывания. В этот момент регистры CS и IP указывают на следующую за HLT команду. При возникновении прерывания, как обычно, в стек записывается содержимое этих регистров и управление передается в операционную систему на соответствующую процедуру обработки прерывания. По завершении ее работы в регистрах CS и IP будет восстановлен адрес команды, следующей за HLT. С этой следующей команды и возобновится выполнение приостановленной программы.

Команда HLT используется, когда процессору нечего делать и он ожидает, например, нажатия какой-нибудь клавишы на клавиатуре.

```
Блокировка шины (lock the bus): LOCK
```

Эта команда блокирует передачу какой-либо информации по линиям связи, пока не будет полностью завершено выполнение следующей за ней команды (перед которой может находиться префикс повторения или префикс сегментного регистра).

Команда LOCK нужна для реализации многозадачного режима работы ПК, когда одновременно выполняется несколько программ. Она используется, чтобы запретить другим программам прерывать текущую программу в тот момент, когда та выполняет сложную операцию (типа пересылки строки из одного места памяти в другое), чтобы другие программы не могли в "середине" такой операции испортить ее операнды.

```
Переключение на сопроцессор (escape): ESC op1,op2
```

По этой команде посылается сигнал арифметическому сопроцессору (8087, 80287 и т. п.), чтобы он выполнил одну из своих операций: op1 задает код этой операции, а op2 указывает местоположение ее операнда. После этого сопроцессор начинает выполнять заказанную операцию, а центральный процессор, не дожидаясь ее окончания, переходит к выполнению команды, следующей за ESC.

Отметим, что в ЯА предусмотрены специальные мнемокоды для разных вариантов команды ESC (например, мнемокод FADD, если нужно выполнить сложение вещественных чисел), поэтому в программах на ЯА в явном виде команда ESC обычно не указывается.

Поскольку сопроцессор работает медленнее центрального процессора (выполняемые сопроцессором операции на вещественными числами требуют значительного времени), то для согласованной (синхронной) работы центральный процессор в общем-то должен приостанавливать свою работу, пока сопроцессор не закончит операцию. Для этого центральный процессор должен использовать следующую команду:

```
Переход в состояние ожидания: WAIT
```

По этой команде центральный процессор прекращает свою работу и ожидает, пока сопроцессор не пришлет сигнал об окончании выполнения заказанной ему операции.

14.3. Дополнительные операторы.

В этом разделе вкратце описываются те операторы ЯА, которые не были рассмотрены в предыдущих главах книги.

```
<константное выражение> SHR <константное выражение>
<константное выражение> SHL <константное выражение>
```

Это операторы сдвига, они относятся к константным выражениям. Оператор SHR (SHL) сдвигает значение своего первого операнда, трактуемое как шкала из 16 бит, вправо (влево) на число разрядов, равное значению второго операнда, которое должно быть неотрицательным целым. Биты, сдвигаемые за пределы шкалы, теряются.

Примеры:

```
N EQU 1011b
  MOV CX,N SHL 2   ;MOV CX,101100b
  AND AH,N SHR 1   ;AND AH,101b
  OR AH,N SHL 5    ;ошибка (2-й операнд – 160h – не байт)
```

Операторы SHR и SHL обычно используются для построения одних масок по другим при работе с упакованными данными, записями.

Действие этих операторов совпадает с действием одноименных команд ПК, однако не надо путать их: операторы вычисляются на этапе трансляции

программы и в машинной программе их уже нет, а команды выполняются на этапе счета программы.

```
LENGTH <имя переменной>
```

Этот оператор относится к константным выражениям. Его операнд должен быть именем переменной, описанной в директиве DB, DW, DD или директиве, определяющей структуру или запись. Значение оператора – коэффициент кратности в конструкции DUP, если она является первым операндом этой директивы, и 1 во всех остальных случаях.

Примеры:

```
A DB 100 DUP(?)            ;LENGTH A = 100
B DW 100 DUP (1,5 DUP(0))  ;LENGTH B = 100
C DD 20,30,66 DUP(0)       ;LENGTH C = 1
D DB 'abcd'                ;LENGTH D = 1
```

Оператор LENGTH имеет смысл, только если в директиве, описывающей имя переменной, указан один операнд и им является конструкция вида k DUP(x); в этом случае оператор сообщает количество (k) элементов, описанных по этой директиве. В остальных случаях смысл оператора малопонятен.

```
SIZE <имя переменной>
```

И этот оператор относится к константным выражениям, а его операнд должен быть таким же, как и в операторе LENGTH. Значение оператора SIZE вычисляется по следующей формуле:

```
SIZE V = (TYPE V) * (LENGTH V)
```

Оператор SIZE имеет смысл, только если в директиве, описывающей имя переменной, указан один операнд и им является конструкция вида k DUP(x); в этом случае оператор указывает количество байтов, занятых всеми элементами, описанных по этой директиве. В остальных случаях оператор мало о чем говорит.

```
.TYPE <имя>
```

Это константный оператор, его операндом может быть любое имя. Значением оператора является число размером в байт, биты которого указывают следующие характеристики имени (нумерация битов ведется справа налево от 0 до 7, значение 1 у бита означает наличие характеристики):

Номер бита	*Характеристика*
0	имя описано как метка или имя процедуры
1	имя описано как переменная
5	имя описано (так или иначе) в программе
7	имя описано (в EXTRN) как внешнее

(остальные биты равны 0).

Для имен регистров, констант, сегментов, групп, макросов, типов записей и структур, полей записей и структур оператор выдает значение 20h. Нулевое значение оператора означает, что имя никак не описано в программе либо является названием команды, директивы или оператора, либо является именем, которому по директиве EQU поставлено в соответствие нечисловое значение.

Оператор .TYPE обычно используется в макроопределениях для проверки имен, заданных как фактические параметры.

```
THIS <тип>
```

Этот оператор относится к адресным выражениям. Его значение – адрес, который равен текущему значению счетчика размещения ($) и которому предписывается указанный тип. (В качестве операнда могут использоваться служебные слова BYTE, WORD, DWORD, NEAR, FAR или соответствующие им числовые значения).

Оператор THIS обычно используется в директиве EQU для порождения имени, адресующего текущую точку программы и имеющую заданный тип (в подобных целях используется и директива LABEL – см. разд. 14.6). Например:

```
  A EQU THIS WORD   ;A и B ссылаются на один и тот же массив,
  B DB 20 DUP (?)   ; но A – массив из слов, а B – из байтов
 L1 EQU THIS FAR    ;L1 и L2 метят одну и ту же команду,
L2: MOV AX,0        ; но L1 – дальняя метка, а L2 – близкая
```

```
HIGH <константное выражение>
LOW <константное выражение>
```

Это константные операторы. Их результат – старший (левый) байт (для HIGH) или младший байт (для LOW) значения операнда. Например:

```
X EQU 1234h
  MOV CL,HIGH X     ;эквивалентно MOV CL,12h
  MOV CL,LOW X      ;эквивалентно MOV CL,34h
```

14.4. Директивы управления листингом

При трансляции программы ассемблер формирует ее листинг, в котором помимо текста на ЯА указываются сгенерированный машинный код, диагностические сообщения об ошибках, таблица имен с их атрибутами и т. д.

Перечисленные ниже директивы влияют на формирование листинга – определяют размер его страниц, указывают, что включать в листинг, а что не включать, и т. п.

```
TITLE <текст>
```

Под текстом (уголки не нужны) здесь понимается последовательность любых символов, начиная с первого символа, отличного от пробела, и до конца строки (если текст большой, от него берутся первые 60 символов). Этот текст становится заголовком, который будет появляться в первой строке

каждой страницы листинга. Директива может быть расположена в любом месте программы. Допустима только одна директива TITLE, появление второй директивы вызовет ошибку. Если директивы нет, начальные строки страниц листинга будут пустыми.

```
SUBTTL <текст>
```

Указанный текст (см. директиву TITLE) становится подзаголовком, который будет появляться во второй строке каждой страницы листинга, начиная с очередной страницы. Если директивы нет, вторая строка страниц остается пустой. В программе может быть любое число директив SUBTTL. (Эта директива может указываться без директивы TITLE.)

```
PAGE [<длина>] [,<ширина>]
PAGE +
PAGE
```

Параметр "длина" – это целое число от 10 до 255 (по умолчанию берется 50), а "ширина" – целое число от 60 до 132 (по умолчанию – 80).

Первый вариант этой директивы устанавливает размер страниц листинга – число строк на странице (длину страницы) и число позиций в каждой строке (ширину). Этот размер устанавливается начиная со страницы, на которой находится директива. Если какой-то из параметров отсутствует, то соответствующий размер страницы не меняется.

Примеры:

```
PAGE 100,60  ;100 строк по 60 позиций в каждой
PAGE ,80     ;80 позиций в строке (число строк не меняется)
```

На каждой странице (в правой части ее первой строки) листинга указывается ее номер в виде S-P, где S – номер секции, а P – номер страницы в секции. Начальный номер равен 1-1. Номер страницы автоматически увеличивается на 1 при переходе на новую страницу, номер же секции не меняется автоматически. Второй вариант директивы (PAGE +) означает переход к новой секции и новой странице листинга, при этом номер секции увеличивается на 1, а номер страницы становится равным 1. (Что считать секцией, когда менять номер секции – решает автор программы.)

Третий вариант директивы означает "насильственный" переход на новую страницу листинга с увеличением номера страницы на 1 и без изменения номера секции.

```
.XLIST
```

По директиве прекращается формирование листинга (сама директива в него не попадает): все последующие строки программы не появятся в листинге.

```
.LIST
```

Директива отменяет действие директивы .XLIST, т. е. восстанавливает формирование листинга (начиная с нее самой). Директива .LIST подразумевается по умолчанию в начале программы.

Пример:

```
.XLIST                ;все предложения включаемого файла
INCLUDE LISTS.ASM     ;LISTS.ASM не записываются в листинг
.LIST
```

```
.SALL
```

После этой директивы в листинг не будут записываться макрорасширения и копии блоков повторений (в листинг попадут только макрокоманды и исходные тексты блоков).

```
.XALL
```

После этой директивы в листинг будут записываться макрорасширения и копии блоков повторений, однако в них будут указываться только те предложения, по которым ассемблер генерирует команды и данные (предложения-комментарии и некоторые директивы типа ASSUME или EQU не попадут в листинг). Этот режим устанавливается по умолчанию в начале программы.

```
.LALL
```

Директива отменяет действие директив .SALL и .XALL, после нее в листинг будут записываться все предложения всех макрорасширений и всех копий блоков повторений.

```
.SFCOND
```

Директива подавляет запись в листинг предложений тех ветвей IF-блоков, которые соответствуют ложным условиям. Этот режим устанавливается по умолчанию в начале программы.

```
.LFCOND
```

Директива отменяет действие директивы .SFCOND, т. е. восстанавливает запись в листинг всех ветвей IF-блоков.

14.5. Директивы контроля за работой ассемблера

Здесь бегло рассматриваются директивы ЯА, позволяющие в той или иной мере контролировать процесс трансляции программы.

14.5.1. Директива %OUT

Встречая при трансляции программы директиву

```
%OUT <текст>
```

ассемблер немедленно выдает на экран с новой его строчки указанный текст (последовательность любых символов до конца строки). Если директива находится внутри макроопределения или блока повторения, то в ее тексте можно указывать формальные параметры макроса или блока (с уточнением, если надо, их границ макрооператором &), которые перед печатью будут замене-

ны на фактические параметры. В машинную программу директива %OUT не попадает.

Такие печати полезны для отслеживания процесса трансляции программы. Например, если имеется макроопределение

```
EX MACRO X
   %OUT Обращение: EX X
   INC X
   ENDM
```

то, скажем, по макрокоманде EX SI будет сформировано макрорасширение

```
%OUT Обращение: EX SI
INC SI
```

по которому ассемблер выдаст на экран текст "Обращение: EX SI", а в машинную программу запишет команду INC SI.

14.5.2. Дополнительные IF-директивы

Недостатком директивы %OUT является то, что она "срабатывает" дважды, поскольку ассемблер два раза просматривает текст транслируемой программы.

Последнее обстоятельство обусловлено трудностями, которые испытывает ассемблер при трансляции ссылок вперед. Дело в том, что если ассемблер, просматривающий текст программы от начала к концу, встретит имя, которое еще не было описано, то он не будет знать, что обозначает это имя, каковы тип и адрес ячейки, помеченной этим именем, и потому не будет знать, как транслировать предложение, в котором встретилось это имя. Чтобы решить проблему со ссылками вперед, ассемблер осуществляет, как говорят, два прохода – он дважды полностью просматривает текст программы. На первом проходе он собирает информацию (типы, адреса и т. п.) обо всех именах, описанных в программе (в том числе и об именах, которые используются в программе до их описания), а на втором проходе, пользуясь этой информацией, уже переводит программу на машинный язык.

Вообще говоря, можно и не знать об этих двух проходах ассемблера, однако иногда это знание полезно и даже необходимо. Например, если по директиве %OUT нужно только один раз выдать текст на экран (только на первом или только на втором проходе), тогда следует воспользоваться следующими IF-директивами условного ассемблирования:

```
IF1
IF2
```

Условие директивы IF1 считается выполненным, когда ассемблер совершает свой первый проход (просматривает текст программы в первый раз), и невыполненным при втором проходе. И наоборот, условие директивы IF2 считается выполненным на втором проходе и невыполненным на первом проходе.

Следующие две директивы также относятся к IF-директивам:

```
IFDEF <имя>
IFNDEF <имя>
```

На первом проходе трансляции условие директивы IFDEF (if defined, если определено) считается выполненным, если указанное имя уже описано (до этой директивы) в программе, и считается невыполненным, если это имя будет описано позже или вообще не описано в программе. На втором проходе условие директивы считается выполненным, если имя вообще описано в программе (до или после этой директивы). В директиве же IFNDEF (if not defined) условие считается выполненным, если имя не было описано до этой директивы (для первого прохода) или если имя вообще не описано в программе (для второго прохода).

Описанными считаются имена, указанные в левой части команд и директив, а также имена полей структур и записей, имена регистров. Названия же команд, директив и операторов не относятся к описанным именам.

Пример. По IF-блоку

```
  IFDEF N
S DB N DUP(?)
  ELSE
S DB 256 DUP(?)
  ENDIF
```

в окончательный текст программы попадет описание S как массива из N байт, если в программе описано имя N, а иначе – как 256-байтового массива.

14.5.3. Условная генерация ошибок

При трансляции программы может возникнуть ситуация (например, в макрокоманде не указан обязательный параметр), когда с точки зрения правил языка в программе все в порядке, но вот с точки зрения логики программы возникла ошибочная ситуация, которую может установить только автор программы. Для того чтобы фиксировать такие ошибки, в ЯА введены директивы условной генерации ошибок. Каждая из них проверяет некоторое условие и, если оно выполнено, записывает в листинг (в ту строчку, где встретилась эта директива) сообщение о принудительной ошибке (forced error). После этого трансляция программы будет продолжена, однако файл с объектным кодом, как и при любой иной ошибке периода трансляции, уже формироваться не будет.

Ниже для каждой директивы условной генерации ошибки приводятся номер ошибки и текст записываемого в листинг сообщения об ошибке.

```
.ERR1      87 Forced error - pass 1
.ERR2      88 Forced error - pass 2
.ERR       89 Forced error
```

Директива .ERR1 генерирует ошибку, если директива встретилась ассемблеру на его первом проходе, директива .ERR2 генерирует ошибку, если она встретилась на втором проходе, а директива .ERR генерирует ошибку на любом проходе.

Например:

```
      IFNDEF X ;если имя X не описано и
      .ERR2    ;если это уже 2-й проход,
      ENDIF    ;то зафиксировать ошибку
```

```
.ERRE <выражение>   90 Forced error - expression equals 0
.ERRNZ <выражение>  91 Forced error - expression not equals 0
```

Директива .ERRE генерирует ошибку, если значение выражения равно 0, а директива .ERRNZ – если значение не равно 0. Выражение должно быть константным и не содержать внешних имен и ссылок вперед. Например:

```
      .ERRE TYPE L - NEAR ;ошибка, если L - близкая метка
```

```
.ERRNDEF <имя>      92 Forced error - symbol not defined
.ERRDEF <имя>       93 Forced error - symbol defined
```

Директива .ERRNDEF генерирует ошибку, если указанное имя еще (до директивы) не описано, а директива .ERRDEF – если имя уже описано. Если это имя является ссылкой вперед, то на первом проходе трансляции оно считается неописанным, а на втором проходе – описанным. Например:

```
      .ERRNDEF N  ;ошибка, если имя N еще не описано
      Y DB N DUP(?)
```

```
.ERRB <текст>              94 Forced error - string blank
.ERRNB <текст>             95 Forced error - string not blank
.ERRIDN <текст1>,<текст2>  96 Forced error - strings identical
.ERRDIF <текст1>,<текст2>  97 Forced error - strings different
```

Директива .ERRB генерирует ошибку, если указанный текст пустой, а директива .ERRNB – если текст непустой. Директива .ERRIDN генерирует ошибку, если указанные тексты совпадают, а директива .ERRDIF – если тексты различаются. Здесь под текстом понимается последовательность символов, заключенная в угловые скобки. Если любая из этих директив находится в макроопределении или блоке повторений, то в текстах можно указывать формальные параметры макроса или блока (при этом допускается использование макрооператора &), которые перед сравнением будут заменены на соответствующие фактические параметры. Например:

```
M MACRO X,Y
  .ERRB <X>        ;;ошибка, если 1-й фактич. параметр опущен
  .ERRIDN <Y>,<CS> ;;ошибка, если 2-й фактич. параметр - это CS
  ...
  ENDM
```

14.6. Дополнительные директивы

В данном разделе вкратце описываются директивы ЯА, которые не были рассмотрены в предыдущих частях книги.

14.6.1. Указание типа процессора и набора команд

Как уже отмечалось, базовой системой команд процессоров фирмы Intel является набор команд процессора 8086 (8088): любая программа, использующая только эти команды, может быть без изменений выполнена и на любом другом процессоре. В то же время в каждой старшей модели имеются дополнительные команды. Например, в процессоре 80186 появились новые команды PUSHA, POPA и др., а также были расширены возможности команд MUL, PUSH, SHR и др. В процессоре 80286 допускаются все команды процессора 80186 и введено несколько новых команд, которые, правда, разрешено использовать только в привилегированном (защищенном) режиме. В следующих процессорах также появились новые команды.

В самих процессорах фирмы Intel нет команд вещественной арифметики, и операции над вещественными числами реализуются либо программным путем (для каждой операции составляется своя процедура), либо за счет подключения арифметического сопроцессора (8087, 80287 и т. д.), который аппаратным способом реализует арифметические операции над вещественными числами. В последнем случае центральный процессор обращается к сопроцессору с помощью команды ESC (см. разд. 14.2), в которой указывается, какую операцию и над какими данными должен выполнить сопроцессор. При этом в ЯА для разных вариантов этой команды введены свои мнемонические обозначения, которые можно рассматривать как "команды сопроцессора". Отметим, что у сопроцессоров также соблюдается преемственность: более старшие модели могут выполнять все операции более младших моделей.

По умолчанию макроассемблер MASM (версия 4.0) допускает использование в программе только команд процессора 8086, применение же иных команд рассматривается как ошибка. Если программа рассчитана на работу с другим процессором или на использование команд сопроцессора, то это надо явно указать ассемблеру, для чего используются указанные ниже директивы. Их можно помещать в любом месте программы, они начинает действовать немедленно и до следующей подобной директивы.

.8086 – допускается использование только команд процессора 8086 (эта директива подразумевается по умолчанию).

.186 – допускается использование всех команд процессора 8086, а также всех новых команд процессора 80186.

.286C – допускается использование всех команд, поддерживаемых процессором 80286 в непривилегированном режиме (их набор полностью совпадает с системой командами процессора 80186).

.286P – допускается использование всех команд процессора 80286, в том числе и команд привилегированного режима.

.8087 – помимо команд основного процессора допускается и использование команд сопроцессора 8087.

.287 – помимо команд основного процессора допускается и использование команд сопроцессора 80287.

Замечание. В версии 4.0 языка MASM не предусмотрены аналогичные директивы для процессора 386 и старше и для сопроцессора 387 и старше, т. к. эта версия была разработана до их появления.

14.6.2. Группы сегментов

Директива

```
<имя группы> GROUP <имя сегмента> {,<имя сегмента>}
```

объединяет перечисленные сегменты в одну группу. Это значит, что все имена из всех этих сегментов будут сегментироваться по одному и тому же сегментному регистру. По какому именно – определяется директивой ASSUME: если в ней указан операнд

```
sr:<имя группы>
```

где sr – какой-то сегментный регистр, то каждое имя из этих сегментов ассемблер будет заменять на адресную пару sr:ofs, где ofs – смещение имени, отсчитанное от начала группы (загрузку этого начала в регистр sr должна осуществить сама программа). Такая замена производится, даже если для какого-то из сегментов группы указан "свой" регистр.

Отметим, что объединение сегментов в группу означает их сегментирование по одному регистру, но не их непрерывное расположение в памяти: сегменты одной группы необязательно будут размещены в памяти подряд, между ними могут оказаться другие сегменты (при размещении сегментов в памяти учитываются их классы, а не их принадлежность к одной группе – см. директиву SEGMENT). Но в любом случае расстояние от начала группы до последнего занятого байта последнего сегмента группы не должно превосходить 64 Кб.

Имя группы должно быть уникальным в программе, а сегменты группы могут быть описаны в тексте программы как до, так и после директивы GROUP. Имя группы относится к константным выражениям, его значение вычисляется аналогично значению имени сегмента.

Пример:

```
  GR GROUP S1,S2
  S1 SEGMENT          ;смещение S1 относительно GR равно 0
   A DB 20h DUP(0)    ;смещение A относительно S1 равно 0
  S1 ENDS
  S2 SEGMENT          ;смещение S2 относительно GR равно 20h
   B DW 1             ;смещение B относительно S2 равно 0
  S2 ENDS
CODE SEGMENT
     ASSUME CS:CODE, ES:GR
     MOV AX,GR
     MOV ES,AX        ;ES=GR
     MOV DH,A         ;эквивалентно MOV DH,ES:0
     MOV CX,B         ;эквивалентно MOV CX,ES:20h
     ...
```

Отметим непоследовательность ЯА при работе с группами. Если во всех командах имена из сегментов группы заменяются на смещения имен, отсчитанные от начала группы, то значением оператора OFFSET N является смещение имени N относительно сегмента, в котором оно описано, даже если этот сегмент входит в группу (так, в приведенном примере значение OFFSET B будет равно 0). Чтобы смещение имени N в этом операторе отсчитывалось от начала группы, надо вместо N записать конструкцию <имя группы>:N (в нашем примере значение оператора OFFSET GR:B равно 20h). Аналогичная проблема возникает при описании адресных констант в директивах DW и DD: если в качестве их операнда указать просто имя, то оно будет заменено ассемблером на смещение имени, отсчитанное от начала того сегмента, где имя описано, а не от начала группы, в которую входит данный сегмент. Здесь также, если нужен отсчет смещения от начала группы, следует применить конструкцию вида <имя группы>:<имя>.

14.6.3. Изменение счетчика размещения

Транслируя программу, ассемблер следит за адресом ее очередного предложения, который он хранит в счетчике размещения и значение которого можно узнать в программе с помощью символа $. Следующие две директивы позволяют менять значение этого счетчика.

```
EVEN
```

Эта директива выравнивает счетчик размещения на ближайший четный адрес: если текущее значение счетчика четно, то оно не меняется, а иначе в очередной байт памяти ассемблер записывает величину 90h (команда NOP) и увеличивает значение счетчика на 1.

```
ORG <выражение>
```

Выражение может быть константным или адресным, но все используемые в нем имена должны быть из текущего сегмента, причем они должны быть описаны до этой директивы. Кроме того, элементом выражения может быть обозначение счетчика размещения ($). Значение этого выражения становится новым значением счетчика размещения; это означает, что очередное предложение программы будет размещено в памяти уже с этого нового адреса (в пропускаемые байты ничего не записывается).

Примеры:

```
ORG 100h
MOV AX,0   ;эта команда будет размещена с адреса 100h
           ;текущего сегмента
ORG $+20   ;пропустить 20 байт
```

Если по директиве ORG значение счетчика размещения уменьшается, тогда очередное предложение "накладывается" на прежнюю величину, помещенную ранее по этому адресу.

14.6.4. Другие директивы

<имя> LABEL <тип>

Эта директива эквивалентна предложению

```
<имя> EQU THIS <тип>
```

(см. разд. 14.3) и определяет имя, которому приписывается текущий адрес (значение счетчика размещения $) и указанный тип. Имя должно быть уникальным в программе. Допустимые типы: BYTE, WORD, DWORD, NEAR, FAR.

Директива LABEL используется для того, чтобы на следующую за ней команду или переменную можно было ссылаться по имени с иным типом, чем это разрешается согласно описанию команды или переменной. Например:

```
 FL LABEL FAR   ;FL и NL метят одну и ту же команду,
NL: MOV AX,0    ;но FL – дальняя метка, а NL – близкая
```

.RADIX <константное выражение>

В ЯА числа могут быть записаны в системах счисления с основаниями 10, 16, 8 и 2 (10d, 10h, 10q, 10b и т. п.). Если буква-спецификатор в конце числа не указана (например, 10), то подразумевается десятичное число. Однако это правило можно изменить с помощью директивы .RADIX: значение ее операнда указывает основание той системы счисления, которая теперь будет подразумеваться, когда число записано без буквы-спецификатора. Значением операнда должно быть число 10, 16, 8 или 2 (если оно указано без спецификатора, то всегда трактуется как десятичное).

Отметим, что если по умолчанию установлена шестнадцатеричная система, то буквы b и d в конце числа трактуются как спецификатор, а не как цифра: например, 10b – это 10 в двоичной системе, а 10d – это 10 в десятичной системе; шестнадцатеричные же числа 10B и 10D надо записывать как 10Bh и 10Dh.

Примеры:

```
.RADIX 16
MOV AX,10       ;AX:=10h=16
MOV AX,10d      ;AX:=10
MOV AX,10Dh     ;AX:=10Dh=269
.RADIX 10       ;десятичное 10 (!)
MOV AX,10       ;AX:=10
MOV AX,10h      ;AX:=10h=16
```

NAME <имя модуля>

По этой директиве указанное имя (от него берется 6 первых символов) присваивается текущему модулю программы и используется компоновщиком в диагностических сообщениях в случае обнаружения ошибок при объединении модулей программы: это имя указывается в скобках после имени файла с данным модулем. (В других целях это имя не используется.)

Например, если в модуле имеется директива NAME MOD1 и если в оттранслированном виде модуль находится в файле PROG.OBJ, тогда возможно такое сообщение об ошибке на этапе компоновки (об имени DDD, указанном в данном модуле как внешнее, но не описанном как общее в других модулях):

```
Unresolved externals: DDD in file(s) PROG.OBJ(MOD1)
```

В модуле должно быть не более одной директивы NAME. Если ее нет, в качестве имени модуля берутся первые 6 символов из текста, указанного в директиве TITLE, а если и ее нет, модулю дается имя A.

СПИСОК ЛИТЕРАТУРЫ

1. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM: Пер. в англ. М.: Радио и связь, 1988. 448 с.
2. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: Пер. с англ. М.: Радио и связь, 1991. 336 с.
3. Абель П. Язык Ассемблера для IBM PC и программирования: Пер. с англ. М.: Высшая школа, 1992. 477 с.
4. Нортон П., Сохуэ Д. Язык ассемблера для IBM PC: Пер. с англ. М.: Компьютер; Финансы и статистика, 1992. 352 с.
5. Использование Turbo Assembler при разработке программ. Киев: "Диалектика", 1994. 288 с.
6. Лямин Л. В. Макроассемблер MASM. М.: Радио и связь, 1994. 320 с.

ОГЛАВЛЕНИЕ

Уважаемые читатели!

Если Вас заинтересовала какая-либо книга нашего издательства, то Вы можете сделать на нее заказ, и мы вышлем ее почтой на условиях ***ПРЕДОПЛАТЫ***.

Обратите внимание – стоимость книг с доставкой ***ниже***, чем в книжном магазине (см. прайс-лист www.dialog-mifi.ru).

Оформление заказа

Для получения книг Вам нужно отправить заявку одним из способов: на e-mail: **zakaz@dialog-mifi.ru** или **dialog_mephi@mail.ru**; почтой по адресу: **115409, г. Москва, ул. Москворечье, д. 31, корп. 2**, Издательство Диалог-МИФИ (***не забудьте вложить конверт с Вашим обратным адресом***).

В заявке обязательно должно быть указано:

- ▶ Фамилия, имя, отчество (полностью);
- ▶ Полный почтовый адрес с индексом;
- ▶ Наименование книги;
- ▶ Количество экземпляров;
- ▶ Телефон и e-mail указываются при наличии.

При получении заявки мы сообщим Вам о наличии книг, номер заказа и сумму оплаты. Только после получения нашего сообщения Вы должны оплатить заказ.

Наши реквизиты:

Название организации	ООО "Издательство Диалог-МИФИ"
Фактический адрес	115409, г. Москва, ул. Москворечье, д. 31, корп. 2
ИНН	7724582966
Расчетный счет (рублевый)	40702810600000003090
Наименование банка	АБ «Интерпрогрессбанк» (ЗАО)
БИК	044525402
Корр/счет	30101810100000000402
Адрес банка	г. Москва, Старо-Каширское шоссе, д. 2, корп. 8

При оформлении бланка оплаты заказа в графе Назначение платежа должно быть указано: "Оплата за заказ №… и Ф. И. О."

Получение книг

Книги отправляются в течение трех рабочих дней со дня поступления денег за заказ на расчетный счет издательства.

Наш телефон: 8-905-769-16-61