Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра ЭВМ

И.В. Лукьянова

Конспект лекций по курсу «Конструирование программ и языки программирования» часть 2 Язык программирования ассемблер

Учебное пособие

для студентов специальности 40 02 01

«Вычислительные машины, системы и сети»

дневной, заочной и дистанционной форм обучения

Минск – 2006

Оглавление

Введение

Язык ассемблера относится к языкам программирования низкого уровня. Споры о том, какой из языков программирования лучше, велись программистами во все времена, и во все времена не могли сойтись во мнениях. На самом деле все языки высокого уровня должны транслировать свои команды в машинный код, т.е. в сущности все языки программирования – это один и тот же язык. Не имеет значения, какой из языков высокого уровня вы предпочитаете, изучите язык ассемблера, только он позволяет разговаривать с компьютером на его собственном языке и непосредственно управлять аппаратными средствами, при этом необходимо вникать в детали и отвечать за свои действия.

Изучение языка ассемблера является необходимой частью подготовки профессиональных программистов, поскольку позволяет шире понять принципы работы ЭВМ, операционных систем и трансляторов с языков высокого уровня.

В данном пособии описана архитектура и системы команд процессора Intel 8086, что является базой для изучения программирования других, более современных, процессоров Intel. Кроме того, в учебном пособии рассмотрены принципы объединения программ на языках высокого уровня и на языке Ассемблера, что является актуальным.

1. Архитектура процессора 8086

Системный блок персонального компьютера содержит: блок питания; системную (материнскую) плату; адаптеры внешних устройств; накопители на жестких магнитных (НЖМД) и гибких (НГМД) дисках, а также ряд других устройств. Для нас наибольший интерес представляет системная плата, на которой размещаются постоянное запоминающее устройство ПЗУ (ROM - read only memory), оперативное запоминающее устройство ОЗУ (RAM - random access memory), процессор и логика управления, связанные между собой шинами.

Физически и ОЗУ и ПЗУ выполнены в виде микросхем. Характерным для персонального компьютера является тот факт, что при выключении электропитания содержимое ОЗУ утрачивается (энергозависимая память), а ПЗУ – нет (энергонезависимая память).

Одна из основных задач ПЗУ обеспечить процедуру старта персонального компьютера. В ПЗУ хранятся базовая система ввода/вывода BIOS, а также некоторые служебные программы и таблицы, например, начальный загрузчик, программа тестирования POST и т.п.

Оперативная память ОЗУ предназначена для временного хранения программ и данных, которыми они манипулируют. Логически

оперативную память можно представить в виде последовательности ячеек, каждая из которых имеет свой номер, называемый адресом.

Центральный процессор (ЦП) в современных персональных компьютерах выполнен в виде одной сверхбольшой интегральной микросхемы (СБИС). ЦП выполняет машинные команды, выбирая их в заданной последовательности из оперативной памяти. Работа всех электронных устройств компьютера координируется сигналами управления, вырабатываемыми ЦП и некоторыми другими СБИС, сигналами тактового генератора, с помощью которых синхронизируются действия всех сигналов.

Возможности компьютера в большей степени зависят от типа установленного процессора и его тактовой частоты. Семейство процессоров 80x86 корпорации Intel включает в себя микросхемы: 8086, 80186, 80286, 80386, 80486, Pentium, Pentium II, Pentium III Pentium IV и т.д. Совместимые с 80x86 микросхемы выпускают также фирмы AMD, IBM, Cyrix. Особенностью этих процессоров является преемственность на уровне машинных команд: программы, написанные для младших моделей процессоров, без каких-либо 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.

Структуру центрального процессора Intel 8086 можно разделить на два логических блока (рис.1.1):

- блок исполнения (EU:Execution Unit);
- блок интерфейса шин (BIU:Bus Interface Unit).

(Интерфейс (interface) - это совокупность средств, обеспечивающих сопряжение устройств и программных модулей как на физическом, так и на логическом уровнях).

В состав EU входят: арифметическо-логическое устройство ALU, устройство управления CU (Control Unit) и десять регистров. Устройства блока EU обеспечивают обработку команд, выполнение арифметических и логических операций.

Три части блока BIU - устройство управления шинами, блок очереди команд и регистры сегментов – предназначены для выполнения следующих функций:

- управление обменом данными с EU, памятью и внешними устройствами ввода/вывода;
- адресация памяти;
- выборка команд (осуществляется с помощью блока очереди команд Queue, который позволяет выбирать команды с упреждением).

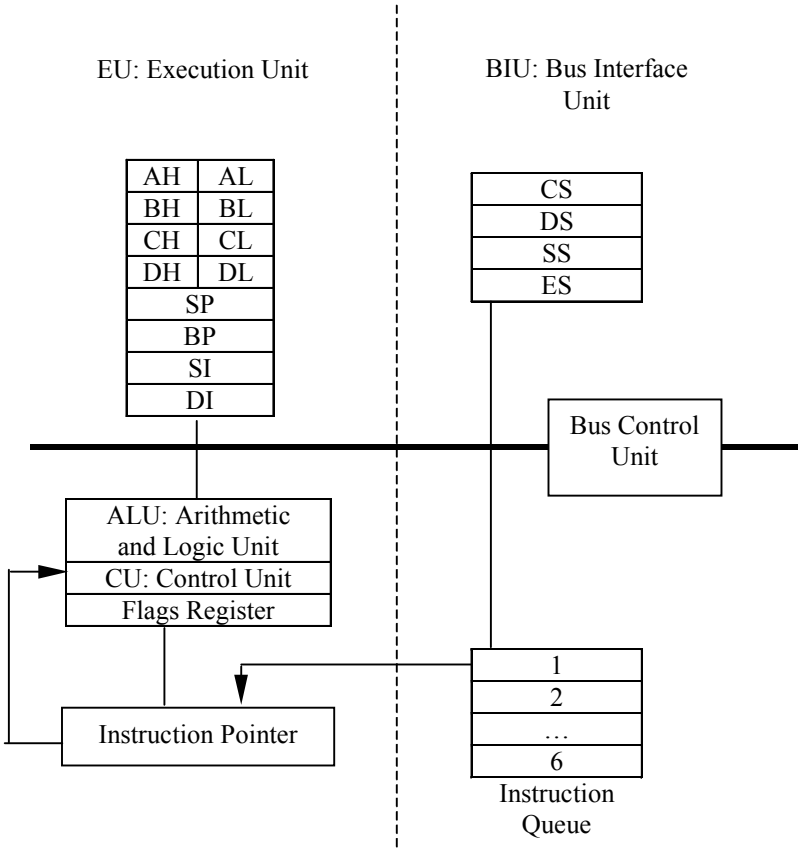

Рис.1.1. Структура центрального процессора.

Три части блока BIU - устройство управления шинами, блок очереди команд и регистры сегментов – предназначены для выполнения следующих функций:

– управление обменом данными с EU, памятью и внешними устройствами ввода/вывода;

– адресация памяти;

– выборка команд (осуществляется с помощью блока очереди команд Queue, который позволяет выбирать команды с упреждением).

С точки зрения программиста, процессор 8086 состоит из 8 регистров общего назначения, 4 сегментных регистров, регистра адреса команд (счетчика команд) и регистра флагов. Процессор выставляет на шину адреса адрес выбираемых из памяти команд (или данных), которые поступают в шестибайтный буфер (очередь команд), а затем исполняются.

Адресную шину можно представить в виде 20 проводников, в каждом из которых может либо протекать напряжение заданного уровня (сигнал 1), либо отсутствовать (сигнал 0). Таким образом, микропроцессор оперирует с двоичной системой счисления (двоичной системой представления данных). Символьная информация кодируется в соответствии с кодом ASCII (Американский стандартный код для обмена информацией). Числовые данные кодируются в соответствии с двоичной

арифметикой. Отрицательные числа представляются в дополнительном коде.

Минимальная единица информации, соответствующая двоичному разряду, называется бит (Bit). Группа из восьми битов называется байтом (Byte) и представляет собой наименьшую адресуемую единицу – ячейку памяти. Биты в байте нумеруют справа налево цифрами 0...7.

BYTE

7	6	5	4	3	2	1	0

Двухбайтовое поле образует шестнадцатиразрядное машинное слово (Word), биты в котором нумеруются от 0 до 15 справа налево. Байт с меньшим адресом считается младшим.

WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Четырехбайтовое поле образует двойное слово (Double Word), а шестнадцатибайтовое – параграф (Paragraph).

Таким образом, с помощью 16-разрядной шины данных можно передавать числа от 0 (во всех проводниках сигнал 0) до 65535 (во всех проводниках сигнал 1). Несмотря на то, что двоичная система обладает высокой наглядностью, она имеет существенный недостаток – числа, записанные в двоичной системе, слишком громоздки. С другой стороны, привычная для нас десятичная система слишком сложна для перевода чисел в двоичную систему счисления и обратно. Поэтому наибольшее распространение в практике программирования получила шестнадцатеричная система счисления.

При написании программ принято двоичные числа сопровождать латинской буквой B или b, (например, 101B), а шестнадцатеричные – буквой H или h на конце. Если число начинается с буквы, то впереди обязательно приписывается нуль, например, 0BA8H.

1.1. Программно доступные регистры микропроцессора

В состав процессора 8086 входит 14 шестнадцатиразрядных регистров, которые используются для управления выполнением команд, адресации операндов и выполнения арифметических и логических операций. Процессоры 80386, 80486 и Pentium могут использовать 32-битные регистры и кроме того имеют дополнительные регистры и расширения, рассмотрение которых мы опустим.

Все регистры можно разделить на следующие группы:

Регистры общего назначения. К ним относятся 16-разрядные регистры AX, BX, CX, DX.

В общем случае функция, выполняемая тем или иным регистром, определяется командами, в которых он используется. При этом с каждым

регистром связано некоторое стандартное его значение. Ниже перечисляются наиболее характерные функции каждого регистра:

Регистр AX или аккумулятор автоматически применяется в операциях умножения, деления, и при работе с портами ввода-вывода (команды in, out).

Регистр BX или регистр базы может содержать адреса элементов в оперативной памяти, которые по умолчанию представляют собой смещение в сегменте данных.

Регистр CX или счетчик используется в различных операциях для хранения числа повторений, например в циклах, в строковых командах и т.п.

Регистр DX или регистр данных применяется в операциях умножения и деления в качестве расширителя аккумулятора.

Регистры AX, BX, CX, и DX позволяют независимо обращаться к их старшей и младшей половине. Соответствующие подрегистры являются восьмиразрядными и имеют имена AH, AL, BH, BL, CH, CL, DH, DL.

Адресные регистры. Существуют четыре 16-битовых регистра, которые могут участвовать в адресации операндов. Это регистры: BX, базовый регистр (он же регистр общего назначения), SI, DI и BP.

Регистр SI или регистр индекса источника, как и регистр BX, может содержать адреса элементов в оперативной памяти, которые по умолчанию представляют собой смещение в сегменте данных. При выполнении операций со строками в этом регистре содержится смещение строки источника в сегменте данных.

Регистр DI или регистр индекса приемника тоже может содержать адреса элементов в оперативной памяти, которые по умолчанию представляют собой смещение в сегменте данных. При выполнении операций со строками в этом регистре содержится смещение строки приемника в сегменте дополнительных данных.

Регистр BP или указатель базы может содержать адреса элементов в оперативной памяти, которые по умолчанию представляют собой смещение в сегменте стека.

Регистр указателя стека . Указатель стека SP – это 16-битовый регистр, который используется для записи данных в стек и чтения их из стека. Фактически он содержит смещение в сегменте стека, которое определяет нужное слово памяти. Значение этого регистра автоматически изменяются командами для работы со стеком типа push, pop, pushf, popf, call, ret.

Все перечисленные регистры можно использовать и для хранения данных, если они не нужны для применения по прямому назначению.

Регистр указателя команд. Регистр IP или регистр указатель команд всегда содержит смещение кода следующей выполняемой

команды. Как только некоторая команда начинает выполняться, значение IP увеличивается на ее длину так, что будет адресовать следующую команду. Обычно команды выполняются в естественном порядке, то есть так, как они располагаются в программе. Нарушают естественную последовательность только команды переходов (они начинаются с буквы j и обычно обозначаются как jxx), команды вызова подпрограмм (call), обработчиков прерываний (int) и возврата (ret, iret). Непосредственно содержимое IP нельзя изменить или прочитать. Косвенно загрузить в регистр IP новое значение могут только команды вида jxx, call, int, ret, iret.

Сегментные регистры. К ним относятся CS,DS,SS и ES.

Регистр CS или регистр сегмента кода определяет стартовый адрес сегмента, в который помещается код выполняемой программы. Это единственный сегментный регистр, который нельзя загрузить непосредственно. Косвенно загрузить в CS новое значение могут только команды вида jxx, call, int, ret, iret. Физический адрес команды в памяти для выполняемой программы определяет пара регистров CS и IP и это записывается в виде CS:IP. Аналогичная форма записи используется для указания физического адреса и в других сегментах.

Регистр DS или регистр сегмента данных определяет стартовый адрес сегмента, в который помещаются данные для программы. По умолчанию смещения в сегменте данных задаются в регистрах BX,SI и DI.

Регистр ES или регистр сегмента дополнительных данных определяет стартовый адрес сегмента, в который помещаются дополнительные данные для программы. Например, в случае строковых команд, DS определяет сегмент для строки-источника, а ES – сегмент для строки-приемника.

Регистр SS или регистр сегмента стека определяет стартовый адрес сегмента, в который помещается стек для программы. По умолчанию смещения в сегменте стека задаются в регистрах SP и BP.

Регистр признаков или флагов. Регистр FLAGS включает в себя биты, каждый из которых устанавливается в единичное или нулевое значение при определенных условиях. Эти биты обозначены буквами C, P, A, Z, S, T, I, D, O, причем нулевому биту соответствует буква C. Доступ к данным в этом регистре можно получить с помощью команд pushf и popf. Ниже определим назначение каждого из битов регистра признаков.

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

Бит CF – признак переноса. CF устанавливается в 1 при возникновении переноса или заема из старшего бита результата, а также

фиксирует значение выдвигаемого бита при сдвиге операнда, может быть установлен в ноль командой clc и в единицу командой stc.

Бит PF – признак четности. Устанавливается в 1, если результат последней операции имеет четное число единиц.

Бит AF– признак вспомогательного переноса. Используется для выполнения операций десятичной арифметики, когда данные хранятся в двоично-десятичном коде. Этот бит фиксирует перенос из младшей тетрады в старшую при сложении либо заем при вычитании.

Бит ZF – признак нуля. Устанавливается в 1 при нулевом результате последней операции.

Бит SF – признак знака. ZF=1, если старший бит результата равен 1. То есть он повторяет значение старшего бита результата, который при использовании дополнительного кода определяет знак числа (единица для отрицательных чисел и ноль для положительных).

Бит TF – признак трассировки. Этот бит устанавливается программистом. Если T=1, то по завершению каждой команды возникает прерывание работ МП. Обычно бит T используют в режиме отладки для передачи управления подпрограмме, индицирующей текущее состояние программы (например содержимое внутренних регистров МП).

Бит IF – признак разрешения прерываний. При I=1 МП реагирует на внешние аппаратные прерывания по входу INTR. При I=0 прерывания по входу INTR запрещаются или маскируются. Значение признака I не влияет на восприятие внешних немаскируемых прерываний (вход NMI МП) и на выполнение внутренних или программных прерываний по команде int. Этот признак может быть установлен в ноль командой cli и в единицу командой sti.

Бит DF – признак направления. Используется при обработке блоков данных. Для указания элементов (байтов, слов и т.д.) в каждом блоке используются индексные регистры SI и DI. После обработки очередного элемента МП автоматически изменяет содержимое индексных регистров для выбора следующего элемента. Если D=0, команды, работающие с блоками, увеличивают содержимое индексных регистров для выбора следующего элемента. Если D=1, то команды уменьшают содержимое этих регистров. Этот признак может быть установлен в ноль командой cld и в единицу командой std.

Бит OF – признак переполнения. Устанавливается в единицу в случае переполнения при выполнении операций сложения и вычитания со знаковыми числами, тем самым указывает на потерю старшего бита результата.

X – зарезервированные биты.

Заметим, что флаги младшего байта регистра признаков устанавливаются арифметическими или логическими операциями

процессора. Флаги старшего байта, кроме флага переполнения, отражают состояние микропроцессора и оказывают влияние на выполнение программы. Они устанавливаются и сбрасываются специальными командами. Флаги младшего байта также используются командами условных переходов для изменения порядка выполнения программы.

1.2. Сегментная организация памяти.

Пусть исполняемый модуль некоторой программы загружен в память персонального компьютера. Команды модуля в заданном порядке считываются в микропроцессор и выполняются. При этом они используют данные, которые выбирают из памяти и регистров микропроцессора. Поскольку для адресации памяти МП 8086 использует 16-разрядные адресные регистры, это обеспечивает ему доступ к 65536 (FFFFh) байт или 64К (1К = 1024 байт = 2^{10} байт) основной памяти, в то время как все адресное пространство его памяти составляет 1 Мбайт.

Рассмотрим, как можно адресовать память объемом 1 Мбайт с помощью 16-разрядных регистров МП. Для этого необходимо разделить всю адресуемую память на сегменты объемом 64 Кбайт, причем принято, что каждый сегмент может начинаться только на границе параграфа (16 байт) от начала памяти, то есть по адресам 0, 16, 32, и т.д., или в шестнадцатеричной системе счисления 0, 10, 20, и т.д. Различные сегменты могут перекрываться или полностью совпадать друг с другом. Начальный адрес каждого сегмента должен храниться в одном из сегментных регистров CS,DS,SS, ES. (Все эти регистры шестнадцатиразрядные.) Таким образом стартовыми для сегментов могут быть адреса памяти 00000_{16}, 00010_{16}, 00020_{16}, …, $FFFE0_{16}$, $FFFF0_{16}$. В сегментные регистры записываются значения этих адресов без последнего нуля, т.е. 0000_{16}, 0001_{16}, 0002_{16}, …, $FFFE_{16}$, $FFFF_{16}$. Эти адреса также называются базовыми адресами сегменьов. Для того, чтобы получить значение полного физического адреса начала сегмента к содержимому сегментных регистров достаточно дописать справа шестнадцатеричный ноль. Физический адрес некоторого элемента в памяти (порядковый номер первого байта элемента от начала памяти) определяется суммой значения, заданного в сегментном регистре, со значением, называемым смещением. Смещение определяет порядковый номер первого байта элемента от начала сегмента и может храниться в одном из следующих регистров МП: IP (для кода), BX, SI, DI (для данных), BP,SP (для стека). Суммирование происходит следующим образом: Микропроцессор расширяет содержимое сегментного регистра (базовый адрес), добавляя к нему 4 младших нулевых бита, при этом адрес становится 20-битным (полный адрес сегмента) и прибавляет к младшим 16 битам значение смещения.

Полученный 20-битовый результат будет представлять собой физический или абсолютный адрес ячейки памяти.

Содержимое сегментного регистра	Базовый адрес xxxx	xxxx	xxxx	xxxx	0000	Полный адрес сегмента
+						
Смещение		xxxx	xxxx	xxxx	xxxx	
	xxxx	xxxx	xxxx	xxxx	xxxx	Абсолютный (физический) адрес

Рис. 1.2. Принцип получения абсолютного адреса.

В командах ассемблера можно использовать так называемый префикс замены, который позволяет переопределять заданные по умолчанию сегменты для разных элементов. Значения смещений в регистрах BX, SI, DI, BP, SP можно увеличивать или уменьшать, но при этом учитывать, что память в ПЭВМ имеет кольцевую организацию. Это значит, что после смещения $FFFF_{16}$ следует смещение 0000_{16} и соответственно перед смещением 0000_{16} следует смещение $FFFF_{16}$.

Как правило для каждой программы в памяти выделяются несколько сегментов, имеющих свое назначение. Существуют три основных типа сегментов:

1. сегмент кода, который содержит машинные команды, адресуется регистром CS;
2. сегмент данных, содержит данные, необходимые программе, адресуется регистром DS;
3. сегмент стека, адресуется регистром SS.

При записи команд на языке Ассемблера принято указывать адреса с помощью следующей конструкции:

```
<адрес сегмента>:<смещение>
```

или

```
<сегментный регистр>:<адресное выражение>
```

1.3. Способы адресации.

Большинство команд процессора содержит аргументы, которые принято называть операндами. Существующие способы задания адресов хранения операндов называются способами адресации. Все способы адресации данных можно отнести к одной из следующих групп:

Непосредственная адресация, прямая адресация, регистровая адресация, косвенная регистровая адресация, относительная косвенная регистровая адресация, базовая индексная адресация, неявная адресация.

Непосредственная адресация. Здесь операнд является частью команды. Операнд помещается в последние байты команды, причем младший байт располагается по меньшему адресу, то есть следует первым.

```
mov  AX, 1234h
```

Здесь шестнадцатеричное значение (признаком шестнадцатеричного числа является буква h) 1234 помещается в регистр AX МП. Младший байт операнда в команде будет содержать значение 34, а старший – 12.

Прямая адресация. Здесь смещение данного является частью команды.

```
mov  AX, my_label
```

В регистр AX помещается смещение метки my_label, которое содержится в самой команде.

Регистровая адресация. Операнд находится в одном из регистров общего назначения или в одном из сегментных регистров. Имя регистра задается в самой команде.

```
mov  AX, BX
```

Здесь значение из регистра BX передается в регистр AX.

Косвенная регистровая адресация. Смещение данного находится в одном из регистров BX, SI, DI, BP. Напомним, что для каждого регистра, содержащего смещение, существует сегмент, заданный по умолчанию.

```
mov  AX, [BX]
```

Выполняются следующие действия: читается значение из регистра BX, которое рассматривается как смещение в памяти в сегменте данных, по полученному смещению из памяти считывается слово, которое помещается в регистр AX. Если задан префикс замены CS, команда перепишется в виде:

```
mov  AX, CS:[BX]
```

Здесь выполняются те же действия, но смещение второго операнда берется в сегменте кода.

Относительная косвенная регистровая адресация. Смещение данного вычисляется как сумма смещения в команде и значения в одном из регистров BX, SI, DI, BP.

```
mov  AX, [BX+10]
```

Здесь смещение данного в сегменте DS определяется суммой значений из регистра BX с числом 10. Можно использовать еще две формы записи этой же команды:

```
mov  AX, [BX]+10
mov  AX, 10[BX]
```

Этот способ адресации называют также базовым, если используются базовые регистры BX,BP и индексным, если используются индексные регистры SI, DI.

Базовая индексная адресация. Смещение данного вычисляется как сумма значений в базовом регистре BX или BP, в индексном регистре SI или DI исмещения в команде. В частном случае смещение может отсутствовать. Этот способ удобно использовать для обращения к элементам двумерных массивов.

```
mov  AX, my_array[BX][SI]
```

Здесь my_array можно рассматривать как адрес начала массива.

Неявная адресация. В этом случае адреса объектов неявно задаются кодом операции. Например, в строковых командах неявно используются регистры SI, DI.

Если в инструкциях процессора указываются регистры в квадратных скобках, например MOV AX,A[BX], то в этом случае команда работает с так называемым исполнительным адресом который вычисляется по формуле $A_{ИСП}=(A+[BX]) \bmod 2^{16}$, где [BX] обозначает содержимое регистра BX. То есть процессор, прежде чем выполнить команду, прибавит к адресу A, указанному в команде (см. пример), текущее содержимое BX, получит некоторый новый адрес и из ячейки с этим адресом возьмет второй операнд. Если в результате суммирования получилась сумма большая 65535, то от нее берутся только последние 16 бит (на это указывает mod в приведенной формуле).

Подобная замена адреса из команды на исполнительный адрес называется модификацией адреса, а регистр, участвующий в модификации – регистром-модификатором. В качестве регистра-модификатора можно использовать не любой регистр, а лишь один из следующих: BX, BP, SI и DI.

Анализ реальных программ на языке Ассемблера, позволяет сделать вывод, что в них, в большинстве случаев, указываются адреса лишь из трех областей памяти – сегмента кода, сегмента данных и сегмента стека. Например, в командах перехода всегда указываются адреса других команд, т.е. ссылки на сегмент кода. В командах работающих со стеком указываются адреса из сегмента стека. В остальных же командах (пересылках, арифметических и т.д.) указываются, как правило, адреса из сегмента данных. С учетом этой особенности реальных программ принят ряд соглашений, которые позволяют во многих командах не указывать явно сегментные регистры, а подразумевать их по умолчанию. Для этого необходимо, чтобы начальные адреса сегментов памяти находились в определенных регистрах, а именно: регистр CS должен указывать на начало сегмента кода, регистр DS – на начало сегмента данных, а регистр SS – на начало сегмента стека. Таким образом, адреса переходов всегда

сегментируются по регистру CS; во всех остальных инструкциях, 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 сегментируется по регистру DS; если же адрес модифицируется по регистру BP, то он считается ссылкой в сегмент стека и поэтому по умолчанию сегментируется по регистру SS.

В случае если приходится работать с сегментами памяти, отличными от сегментов кода, стека или данных, в командах, содержащих ссылку на эти сегменты, необходимо явно указывать сегментный регистр. Например:

MOV AX, ES:[BX].

1.4. Организация стека.

Стек – это специально выделенная область оперативной памяти, использующая механизм безадресной записи и выборки элементов данных. Этот механизм предполагает, что элемент, записанный последним, будет всегда прочитан первым. Адрес сегмента памяти, в котором располагается стек, определяется регистром SS. Для МП 8086 данные в стек помещаются только в виде слов (по 2 байта). Пара регистров SS:SP всегда указывает на текущую вершину стека. Для занесения слова в стек используется команда push, при этом сначала смещение в SP уменьшается на два, а затем производится запись слова в стек. Для извлечения слова из стека используется команда pop, при этом сначала производится чтение стека, а затем смещение в SP увеличивается на два. Важно помнить, что стек растет в направлении к началу памяти и уменьшается в направлении к ее концу. Стек предназначен для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний.

1.5. Организация прерываний.

Иногда в процессе работы процессора возникает необходимость приостановки выполнения текущей программы и запуска другой программы – программы-обработчика, специально написанной для обработки некоторой ситуации. Такие особые ситуации называются прерываниями и делятся на два вида: внешние и внутренние прерывания. Внешние прерывания происходят, когда устройство, подключенное к процессору, само генерирует сигнал прерывания. Внутренние прерывания исходят от процессора в двух случаях: в результате выполнения программной команды int или при определенных условиях, таких как деление на ноль при выполнении команды div, когда генерируется сигнал прерывания для обработки этого типа ошибки.

При переходе к процедуре обслуживания прерывания необходимо предусмотреть механизм возврата к прерванной программе. Для этого перед обращением к процедуре прерывания должно быть сохранено состояние всех регистров и флагов, используемых процедурой прерывания, после окончания прерывания эти регистры должны быть восстановлены.

Некоторыми видами прерываний управляют флаги IF и TF, которые для восприятия прерываний должны быть правильно установлены. Если условия для прерывания удовлетворяются и необходимые флаги установлены, то микропроцессор завершает текущую команду, а затем реализует последовательность прерывания:

- текущее значение регистра флагов включается в стек;
- текущее значение кодового сегмента включается в стек;
- текущее значение счетчика команд включается в стек;
- сбрасываются флаги IF и TF.

Новое содержимое счетчика команд и сегмента кода определяют начальный адрес первой команды процедуры обслуживания прерывания. Возврат в прерванную программу осуществляется командой, которая извлекает из стека содержимое для:

- указателя стека;
- сегмента кода;
- регистра флагов.

Адрес программы обслуживания прерывания, который должен быть загружен в счетчик команд и сегмент кода, называется вектором прерывания. Каждому типу прерывания соответствует свой вектор прерывания 0...255, и адрес вектора прерывания находится путем умножения номера типа на четыре (размер вектора прерывания). Таким образом, все вектора прерываний образуют таблицу векторов прерываний, которая содержится в памяти по адресу 0000h:0000h и инициализируется при загрузке компьютера.

2. Загрузка и выполнение программ в DOS

Пусть в оперативную память компьютера загружен выполняемый модуль некоторой программы. Команды программы должны в установленном порядке считываться в микропроцессор и выполняться, при этом они используют данные, расположенные в памяти и регистрах микропроцессора. При загрузке программ в оперативную память DOS (дисковая операционная система) обычно выделяет для программы три сегмента памяти, имеющие самостоятельное назначение. Они содержат код, данные и стек программы. Некоторые сегменты могут и отсутствовать. DOS инициализирует как минимум три сегментных

регистра: CS, DS и SS. При этом совокупности байтов, представляющих команды процессора (код программы), и данные помещаются из файла на диске в оперативную память, а адреса этих сегментов записываются в CS и DS соответственно. Сегмент стека либо выделяется в области, указанной в программе, либо совпадает (если он явно в программе не описан) с самым первым сегментом программы. Адрес сегмента стека помещается в регистр SS. Программа может иметь несколько кодовых сегментов и сегментов данных и в процессе выполнения специальными командами выполнять переключения между ними.

Для того чтобы адресовать одновременно два сегмента данных, например, при выполнении операции пересылки из одной области памяти в другую, можно использовать регистр дополнительного сегмента ES. Кодовый сегмент и сегмент стека всегда определяются содержимым своих регистров (CS и SS), и поэтому в каждый момент выполнения программы всегда используется какой-то один кодовый сегмент и один сегмент стека. Причем если переключение кодового сегмента – довольно простая операция, то переключать сегмент стека можно только при условии четкого представления логики работы программы со стеком, иначе это может привести к зависанию системы.

Все сегменты могут использовать различные области памяти, а могут частично или полностью перекрываться .

Кодовый сегмент должен обязательно описываться в программе, все остальные сегменты могут отсутствовать. В этом случае DOS при загрузке программы в оперативную память инициирует регистры DS и ES значением адреса префикса программного сегмента PSP (Program Segment Prefics) – специальной области оперативной памяти размером 256 (100h) байт. PSP может использоваться в программе для определения имен файлов и параметров из командной строки, введенной при запуске программы на выполнение, объема доступной памяти, переменных окружения системы и т.д. Регистр SS при этом инициализируется значением сегмента, находящегося сразу за PSP, т.е. первого сегмента программы. При этом необходимо учитывать, что стек «растет вниз» (при помещении в стек содержимое регистра SP, указывающего на вершину стека, уменьшается, а при считывании из стека – увеличивается). Таким образом, при помещении в стек каких-либо значений они могут затереть PSP и программы, находящиеся в младших адресах памяти, что может привести к непредсказуемым последствиям. Поэтому рекомендуется всегда явно описывать сегмент стека в тексте программы, задавая ему размер, достаточный для нормальной работы.

После инициализации в регистре IP находится смещение первой команды программы относительно начала кодового сегмента, адрес которого помещен в регистр CS. Процессор, считывая эту команду,

начинает выполнение программы, постоянно изменяя содержимое регистра IP и при необходимости CS для получения кодов очередных команд до тех пор, пока не встретит команду завершения программы. DS после загрузки программы установлен на начало PSP, поэтому для его использования в первых двух командах программы выполняется загрузка DS значением сегмента данных.

```
MOV  AX,DATA
MOV  DS,AX
```

2.1. EXE- и COM-программы

DOS может загружать и выполнять программные файлы двух типов – COM и EXE.

Ввиду сегментации адресного пространства процессора 8086 и того факта, что переходы (JMP) и вызовы (CALL) используют относительную адресацию, оба типа программ могут выполняться в любом месте памяти. Программы никогда не пишутся в предположении, что они будут загружаться с определенного адреса (за исключением некоторых самозагружающихся, защищенных от копирования программ).

Файл COM-формата – это двоичный образ кода и данных программы. Такой файл должен занимать менее 64K и не содержать перемещаемых адресов сегментов.

Файл EXE-формата содержит специальный заголовок, при помощи которого загрузчик выполняет настройку ссылок на сегменты в загруженном модуле.

Перед загрузкой COM- или EXE-программы DOS определяет сегментный адрес, называемый префиксом программного сегмента (PSP), как базовый для программы. Затем DOS выполняет следующие шаги:

– создает копию текущего окружения DOS (область памяти, содержащая ряд строк в формате ASCIIZ, которые могут использоваться приложениями для получения некоторой системной информации и для передачи данных между программами) для программы;

– помещает путь, откуда загружена программа, в конец окружения;

– заполняет поля PSP информацией, полезной для загружаемой программы (количество памяти, доступное программе; сегментный адрес окружения DOS; текущие векторы прерываний INT 22H INT 23H и INT 24H и т.д).

EXE-программы. EXE-программы содержат несколько программных сегментов, включая сегмент кода, данных и стека. EXE-файл загружается, начиная с адреса PSP:0100h. В процессе загрузки считывается информация

заголовка EXE в начале файла и выполняется перемещение адресов сегментов. Это означает, что ссылки типа

```
mov  ax,data_seg
mov  ds,ax
```

и

```
call my_far_proc
```

должны быть приведены (пересчитаны), чтобы учесть тот факт, что программа была загружена в произвольно выбранный сегмент.

После перемещения управление передается загрузочному модулю посредством инструкции далекого перехода (FAR JMP) к адресу CS:IP, извлеченному из заголовка EXE.

В момент получения управления программой EXE -формата:

– DS и ES указывают на начало PSP

– CS, IP, SS и SP инициализированы значениями, указанными в заголовке EXE

– поле PSP MemTop (вершина доступной памяти системы в параграфах) содержит значение, указанное в заголовке EXE. Обычно вся доступная память распределена программе.

COM-программы. COM-программы содержат единственный сегмент (или, во всяком случае, не содержат явных ссылок на другие сегменты). Образ COM-файла считывается с диска и помещается в память, начиная с PSP:0100h. В общем случае, COM-программа может использовать множественные сегменты, но она должна сама вычислять сегментные адреса, используя PSP как базу.

COM-программы предпочтительнее EXE-программ, когда дело касается небольших ассемблерных утилит. Они быстрее загружаются, ибо не требуется перемещения сегментов, и занимают меньше места на диске, поскольку заголовок EXE и сегмент стека отсутствуют в загрузочном модуле.

После загрузки двоичного образа COM-программы:

– CS, DS, ES и SS указывают на PSP;

– SP указывает на конец сегмента PSP (обычно 0FFFEH, но может быть и меньше, если полный 64K сегмент недоступен);

– слово по смещению 06H в PSP (доступные байты в программном сегменте) указывает, какая часть программного сегмента доступна;

– вся память системы за программным сегментом распределена программе;

– слово 00H помещено (PUSH) в стек.

– IP содержит 100H (первый байт модуля) в результате команды JMP PSP:100H.

2.2. Выход из программы

Завершить программу можно следующими способами:

– через функцию 4CH (EXIT) прерывания 21H в любой момент, независимо от значений регистров;

– через функцию 00H прерывания 21H или прерывание INT 20H, когда CS указывает на PSP.

Функция DOS 4CH позволяет возвращать родительскому процессу код выхода, который может быть проверен вызывающей программой или командой COMMAND.COM "IF ERRORLEVEL".

Можно также завершить программу и оставить ее постоянно резидентной (TSR), используя либо INT 27H , либо функцию 31H (KEEP) прерывания 21H. Последний способ имеет те преимущества, что резидентный код может быть длиннее 64K, и что в этом случае можно сформировать код выхода для родительского процесса.

3. Ассемблер, макроассемблер, редактор связей

Существует несколько версий программы ассемблер. Одним из наиболее часто используемых является пакет Turbo Assembler, водящий в состав комплекса программ Borland C++. Рассмотрим работу с этим пакетом более подробно.

Входной информацией для ассемблера (TASM.EXE) является исходный файл — текст программы на языке ассемблера в кодах ASCII. В результате работы ассемблера может получиться до 3-х выходных файлов:

1) объектный файл – представляет собой вариант исходной программы, записанный в машинных командах;

2) листинговый файл – является текстовым файлом в кодах ASCII, включающим как исходную информацию, так и результат работы программы ассемблера;

3) файл перекрестных ссылок – содержит информацию об использовании символов и меток в ассемблерной программе (перед использованием этого файла необходима его обработка программой CREF).

Объектному файлу ассемблер присваивает то же имя, что и у исходного, но с расширением OBJ; для листингового файла и файла перекрестных ссылок принимается значение NUL — специальный тип файла, в котором все, что записывается, недоступно и не может быть восстановлено.

Если ассемблер во время ассемблирования обнаруживает ошибки, он записывает сообщения о них в листинговый файл. Кроме того, он выводит их на экран дисплея.

Программа, полученная в результате ассемблирования (объектный файл), еще не готова к выполнению. Ее необходимо обработать командой редактирования связей TLINK, которая может связать несколько различных объектных модулей в одну программу и на основе объектного модуля формирует исполняемый загрузочный модуль.

Входной информацией для программы TLINK являются имена объектных модулей (файлы можно указывать без расширение OBJ). Если файлов больше одного, то их имена вводятся через разделитель «+». Модули связываются в том же порядке, в каком их имена передаются программе TLINK. Кроме того, TLINK требует указания имени выходного исполняемого модуля. По умолчанию ему присваивается имя первого из объектных модулей, но с расширением EXE. Далее можно указать имя файла, для хранения карты связей (по умолчанию формирование карты не производится). Последнее, что указывается программе TLINK – это библиотеки программ, которые могут быть включены в полученный при связывании модуль. По умолчанию такие библиотеки отсутствуют.

Информацию обо всех этих файлах программа TLINK запрашивает у пользователя после ее вызова.

Графически процесс создания программы на языке Ассемблера можно представить как это показано на рис 3.1.

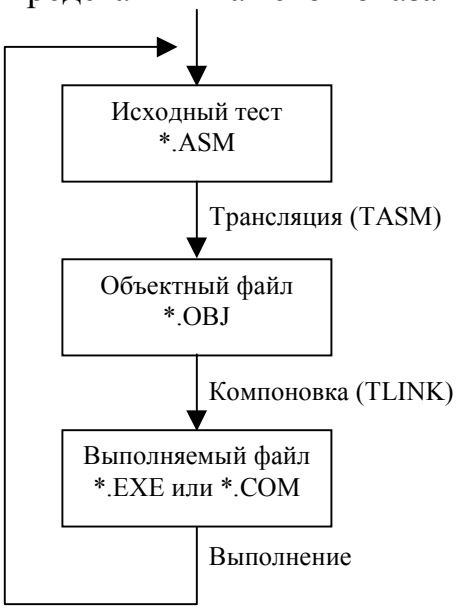

Рис. 3.1. Создание программы на языке Ассемблера.

4. Введение в язык Ассемблера

4.1. Структура программы на языке ассемблера

Предложения и комментарии языка ассемблера представляют собой комбинацию знаков, входящих в алфавит языка, а также чисел и

идентификаторов, которые тоже формируются из знаков алфавита. Алфавит языка составляют цифры, строчные и прописные буквы латинского алфавита, а также следующие символы:

```
? @ _ $ : . [ ] ( ) < > { } + / * & % ! ' ~ | \ = # ^ ; , ` "
```

Одна из возможных структур программы на языке ассемблера выглядит так:

```
.MODEL small
.STACK 300h
.DATA
        ; объявление данных
.CODE
start:          ; или любая другая метка, например, begin
                ; команды программы
END start       ; если выбрана метка begin, то END begin
```

Здесь директива .MODEL позволяет определить используемую модель памяти. В приведенном примере задана модель small, которая предполагает использование двух сегментов памяти (каждый объемом до 64 Кбайт): первый для кода, а второй для данных и стека. Директива .STACK позволяет определить стек размером 300h. Директива .DATA начинает область, в которой объявляются данные. Директива .CODE начинает область, в которой содержится код программы. Метка start: определяет точку, с которой произойдет запуск программы на выполнение (старт программы будет определять метка, записанная после директивы END). Директива END указывает на завершение кода программы. По умолчанию компилятор не различает в ассемблерной программе строчные и прописные буквы.

Предложения языка ассемблера определяют структуру и функции программы, они могут начинаться с любой позиции и содержать не более 128 символов. 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.

Все предложения языка ассемблера делятся на команды и директивы. Общий формат их записи выглядит следующим образом:

<метка> <команда/дректива> <операнды> <;комментарии>

Здесь <метка> - необязательное символическое имя команды или директивы; <команда/дректива> - это обязательный параметр, который представляет собой имя инструкции или директивы; <операнды> - параметр, задающий операнды; <;комментарии> - необязательный параметр, задающий комментарии. Операндов в команде может быть ни одного, один, два и более. Ими могут быть константы, переменные, адреса памяти, имена регистров МП и т.п.

Рассмотрим компоненты ассемблерного предложения.

Метки - это символьные имена, используемые для нахождения в памяти строки, в которой они записаны. Символьные имена задают и переменные и включают следующие знаки: A-Z, a-z, _, @, $, ?, 0-9, причем первый знак не должен быть цифрой, $ и ?. В качестве символьных имен нельзя использовать зарезервированные слова языка ассемблера (ключевые слова). Ключевыми словами ассемблера являются: директивы ассемблера; инструкции процессора; имена регистров; операторы выражений. В идентификаторах одноименные строчные и заглавные буквы считаются эквивалентными. Например, идентификаторы AbS и abS считаются совпадающими. Как правило, каждая метка определяется только один раз, и далее сколько угодно раз может использоваться в качестве операнда. Допустимы строки, содержащие только метки, в этом случае они задают адрес команды или директивы в следующей строке программы. Метки, которые записываются в строке, не содержащей директиву, должны оканчиваться двоеточием.

Команды и директивы определяют действия, которые должны быть выполнены в строке с ассемблерным кодом. Команды или инструкции процессора представляют собой мнемоническую форму записи машинных команд, непосредственно выполняемых микропроцессором. Все инструкции в соответствии с выполняемыми ими функциями делятся на 5 групп:

1) инструкции пересылки данных;
2) арифметические, логические и операции сдвига;
3) операции со строками;
4) инструкции передачи управления;
5) инструкции управления процессором.

Директивы ассемблера действуют лишь в период компиляции программы и позволяют устанавливать режимы компиляции, задавать структуру сегментации программы, определять содержимое полей данных, управлять печатью листинга программы, а также обеспечивают условную компиляцию и некоторые другие функции. В результате обработки директив компилятором объектный код не генерируется.

Операнды определяют непосредственные значения, регистры, фрагменты памяти и другие элементы, которыми манипулируют команды.

Число операндов неявно задается кодом команды. Операндами могут быть имена регистров МП, константы, выражения и метки.

Комментарии используются для документирования программы, они не влияют на размер выполняемого кода.

4.2. Операторы языка ассемблера

Выражения языка ассемблера могут быть использованы в инструкциях или директивах и состоят из операндов и операторов.

Операторы выполняют арифметические, логические, побитовые и другие операции над операндами выражений.

Ниже даны описания наиболее часто используемых в выражениях операторов.

Арифметические операторы.

```
выражение_1    *    выражение_2
выражение_1    /    выражение_2
выражение_1    MOD  выражение_2
выражение_1    +    выражение_2
выражение_1    –    выражение_2
               +    выражение
               –    выражение
```

Эти операторы обеспечивают выполнение основных арифметических действий (здесь MOD - остаток от деления выражения_1 на выражение_2, а знаком / обозначается деление нацело). Результатом арифметического оператора является абсолютное значение.

Операторы сдвига.

```
выражение SHR  счетчик
выражение SHL  счетчик
```

Операторы SHR и SHL сдвигают значение выражения соответственно вправо и влево на число разрядов, определяемое счетчиком. Биты, выдвигаемые за пределы выражения, теряются. Замечание: не следует путать операторы SHR и SHL с одноименными инструкциями процессора.

Операторы отношений.

```
выражение_1    EQ   выражение_2
выражение_1    NE   выражение_2
выражение_1    LT   выражение_2
выражение_1    LE   выражение_2
выражение_1    GT   выражение_2
выражение_1    GE   выражение_2
```

Мнемонические коды отношений расшифровываются следующим образом:

EQ – равно;

NE – не равно;

LT – меньше;

LE – меньше или равно;

GT – больше;

GE – больше или равно.

Операторы отношений формируют значение 0FFFFh при выполнении условия и 0000h в противном случае. Выражения должны иметь абсолютные значения. Операторы отношений обычно используются в директивах условного ассемблирования и инструкциях условного перехода.

Операции с битами.

```
                NOT  выражение
выражение_1     AND  выражение-2
выражение_1     OR   выражение-2
выражение_1     XOR  выражение-2
```

Мнемоники операций расшифровываются следующим образом:

NOT – инверсия;

AND – логическое И;

OR – логическое ИЛИ;

XOR – исключающее логическое ИЛИ.

Операции выполняются над каждыми соответствующими битами выражений. Выражения должны иметь абсолютные значения.

Оператор индекса.

```
[[выражение_1]] [выражение_2]
```

Оператор индекса [] складывает указанные выражения подобно тому, как это делает оператор +, с той разницей, что первое выражение необязательно, при его отсутствии предполагается 0 (двойные квадратные скобки указывают на то, что операнд не обязателен).

Оператор PTR

```
тип  PTR  выражение
```

При помощи оператора PTR переменная или метка, задаваемая выражением, может трактоваться как переменная или метка указанного типа. Тип может быть задан одним из следующих имен или значений:

Таблица 4.1. Типы оператора PTR

Имя типа	Значение
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

Оператор PTR обычно используется для точного определения размера, или расстояния, ссылки. Если PTR не используется, ассемблер подразумевает умалчиваемый тип ссылки. Кроме того, оператор PTR

используется для организации доступа к объекту, который при другом способе вызвал бы генерацию сообщения об ошибке (например, для доступа к старшему байту переменной размера WORD).

Операторы HIGH и LOW

```
HIGH выражение
LOW  выражение
```

Операторы HIGH и LOW вычисляют соответственно старшие и младшие 8 битов значения выражения. Выражение может иметь любое значение.

Оператор SEG

```
SEG  выражение
```

Этот оператор вычисляет значение атрибута СЕГМЕНТ выражения. Выражение может быть меткой, переменной, именем сегмента, именем группы или другим символом.

Оператор OFFSET

```
OFFSET выражение
```

Этот оператор вычисляет значение атрибута СМЕЩЕНИЕ выражения. Выражение может быть меткой, переменной, именем сегмента или другим символом. Для имени сегмента вычисляется смещение от начала этого сегмента до последнего сгенерированного в этом сегменте байта.

Оператор SIZE

```
SIZE переменная
```

Оператор SIZE определяет число байтов памяти, выделенных переменной.

4.3. Приоритеты операций

При вычислении значения выражения операции выполняются в соответствии со следующим списком приоритетов (в порядке убывания):

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

(Некоторые операции не были рассмотрены выше ввиду довольно редкого их использования)

4.4. Объявление и инициализация данных.

Для задания размеров, содержимого и местоположения полей данных, используемых в программе на языке ассемблера, служат директивы определения данных.

Директивы определения данных могут задавать:

- скалярные данные;
- записи, позволяющие манипулировать с данными на уровне бит;
- структуры, отражающие некоторую логическую структуру данных.

4.4.1. Скалярные данные

Для определения области памяти под скалярные данные можно использовать директивы DB, DW, DD, DF, DP, DQ, DT. Они позволяют выделить и инициализировать память следующего типа и размера:

DB – 1 байт для данного;
DW – 2 байта или слово для данного или смещения в памяти;
DD – 4 байта или двойное слово для данного или адреса памяти;
DF, DP – 6-байтный указатель типа far (дальний);
DQ – 8 байт для данного;
DT – 10 байт для данного.
Например:

```
my_byte DB 25 ;  в выделенный байт записано значение 25
my_word DW 1000  ; в выделенное слово записано значение 1000
```

В качестве значения может кодироваться целое число, строковая константа, оператор DUP (см. ниже), абсолютное выражение или знак «?». Знак «?» обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная указанного типа с соответствующим данному значению указателя позиции смещением.

Если в одной директиве определения памяти заданы несколько значений, им распределяются последовательные байты памяти. В этом случае, имя, указанное в начале директивы, именует только первый из этих байтов, остальные остаются безымянными. Для ссылок на них используется выражение вида имя+k, где k – целое число.

Для определения ASCII-кода символа этот символ необходимо заключить в одинарные или двойные кавычки, например

my_byte DB ‘A’ ; в выделенный байт записан ASCII-код буквы A

Строковая константа может содержать столько символов, сколько помещается на одной строке. Символы строки хранятся в памяти в порядке их следования, т.е. 1-й символ имеет самый младший адрес, последний - самый старший.

Во всех директивах определения памяти в качестве одного из значений может быть задан оператор DUP. Он имеет следующий формат:

```
счетчик DUP (значение,...)
```

Указанный в скобках список значений повторяется многократно в соответствии со значением счетчика. Каждое значение в скобках может быть любым выражением, имеющим значением целое число, символьную константу или другой оператор DUP (допускается до 17 уровней вложенности операторов DUP). Значения, если их несколько, должны разделяться запятыми.

Оператор DUP может использоваться не только при определении памяти, но и в других директивах.

Примеры директив определения скалярных данных:

```
integer1  DB   25
string1   DB   'ABCDEff'
empty1    DB   ?
contan2   DW   5*3
string3   DB   'abcd'
high4     DQ   18446744073709551615
high5     DT   120892581961462917470617 5d
db6       DB   5 DUP(5 DUP(5 DUP(10)))
dw6       DW   DUP(1,2,3,4,5)
```

Данные могут быть заданы в десятичной, двоичной, восьмеричной и шестнадцатеричной формах. Каждая форма задается определенным суффиксом в конце числа. Десятичные значения задаются без суффикса или с суффиксом d, двоичные значения задаются с суффиксом b, восьмеричные – с суффиксом o или q, шестнадцатеричные – с суффиксом h и должны начинаться с одной из цифр от 0 до 9.

Например:

```
_bb DB 10011101b
_ww DW 1234o
_dd DD 12345678h
mov ax, 10000
mov bx, 10000d
my_array DW 0,1,2,3,4,5,6,7,8,9
```

Последняя строка выделяет десять слов памяти и записывает в них значения от 0 до 9. Метка my_array определяет смещение начала этой области в сегменте .DATA. Допускается инициализация блоков памяти одними и теми же значениями.

```
Block_array DW 100 DUP (12h)
```

Здесь выделяется память размером 100 слов, в каждое слово помещается значение12h, и метка Block_array хранит смещение этой области в сегменте .DATA.

sym_array DB 10 DUP (“5”)

my_string DB ‘a’,’b’,’c’,’d’,’e’,’f’,0dh,0ah,’$’

Или

my_string DB ‘abcdef’,0dh,0ah,’$’

В качестве начальных значений элементов данных наряду с константами можно использовать выражения и метки, например

my_exp DB –((5*4)/2+1) ; в байте будет задано значение –11.

В тех случаях, когда необходимо выделить память, но не инициализировать ее, используется знак ?, например

my_example DW 20 DUP (?)

no_init DD ?

4.4.2. Записи

Запись представляет собой набор полей бит, объединенных одним именем. Каждое поле записи имеет собственную длину, исчисляемую в битах, и не обязано занимает целое число байтов. Объявление записи в программе на языке ассемблера включает в себя 2 действия:

- объявление шаблона или типа записи директивой RECORD;
- объявление собственно записи.

Формат директивы RECORD:

```
имя_записи     RECORD    имя_поля:длина[[=выражение]],...
```

Директива RECORD определяет вид 8- или 16-битовой записи, содержащей одно или несколько полей. Имя_записи представляет собой имя типа записи, которое будет использоваться при объявлении записи. Имя_поля и длина (в битах) описывает конкретное поле записи. Выражение, если оно указано задает начальное (умалчиваемое) значение поля. Описания полей записи в директиве RECORD, если их несколько, должны разделяться запятыми. Для одной записи может быть задано любое число полей, но их суммарная длина не должна превышать 16 бит.

Длина каждого поля задается константой в пределах от 1 до 16. Если общая длина полей превышает 8 бит, ассемблер выделяет под запись 2 байта, в противном случае – 1 байт. Если задано выражение, оно определяет начальное значение поля. Если длина поля не меньше 7 бит, в качестве выражения может быть использован символ в коде ASCII. Выражение не должно содержать ссылок вперед. Пример:

```
item RECORD    char:7='Q',weight:4=2
```

Запись item будет иметь следующий вид:

	char	weight
0000	1010001	0010

При обработке директивы RECORD формируется шаблон записи, а сами данные создаются при объявлении записи, которое имеет следующий вид:

```
[[имя]]   имя_записи     <[[значение,...]]>
```

По такому объявлению создается переменная типа записи с 8- или 16-битовым значением и структурой полей, соответствующей шаблону, заданному директивой RECORD с именем имя_записи. Имя задает имя переменной типа записи. Если имя опущено, ассемблер распределяет память, но не создает переменную, которую можно было бы использовать для доступа к записи.

В скобках <> указывается список значений полей записи. Значения в списке, если их несколько, должны разделяться запятыми. Каждое значение может быть целым числом, строковой константой или выражением и должно соответствовать длине данного поля. Для каждого поля может быть задано одно значение. Скобки <> обязательны, даже если начальные значения не заданы. Пример:

```
table     item 10 DUP(<'A',2>)
```

Если для описания шаблона записи использовалась директива RECORD из предыдущего примера, то по этому объявлению создается 10 записей, объединенных именем table.

4.4.3. Структуры

Структура представляет собой набор полей байтов, объединенных одним именем. Объявление структуры, аналогично объявлению записи, включает в себя 2 действия:

- объявление шаблона или типа структуры;
- объявление собственно структуры.

Формат объявления типа структуры:

```
имя STRUC
описания_полей
имя ENDS
```

Директивы STRUC и ENDS обозначают соответственно начало и конец описания шаблона (типа) структуры. Описание типа структуры задает имя типа структуры и число, типы и начальные значения полей структуры. Описания_полей определяют поля структуры и могут быть заданы аналогично описанию скалярных типов. Пример:

```
table     STRUC
     count     DB 10
     value     DW 10 DUP(?)
     tname     DB 'font'
table     ENDS
```

При обработке директив STRUC и ENDS формируется шаблон структуры, а сами данные создаются при объявлении структуры, которое имеет следующий вид:

```
[[имя]]   имя_структуры  <[[значение,...]]>
```

Значения полей в объявлении структуры аналогично значению полей при объявлении записи.

4.4.4. Директива эквивалентности

Константы в языке Ассемблера описываются с помощью директивы эквивалентности EQU, имеющей следующий синтаксис:

```
<имя>      EQU  <операнд>
```

Здесь обязательно должно быть указано имя и только один 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. Директиву EQU можно ставить в любое место программы.

4.5. Простейшие директивы определения сегментов

Простейшими директивами определения сегментов являются **.MODEL, .STACK, .DATA, .CODE, .DATA?, .FARDATA, .FARDATA?, .CONST, .DOSSEG.**

Директива **.MODEL** позволяет задавать модель памяти для ассемблерной программы. Она должна следовать перед директивами определения сегментов типа **.STACK, .DATA, .CODE .** Каждая модель фиксирует определенный способ адресации данных и команд программы. Различают ближний (near) и дальний (far) способы адресации. При выборе первого способа значение сегментного адреса данных или кода не изменяется, поэтому изменение адреса сводится только к изменению смещения. Для дальнего способа адресациии изменение адреса сводится к изменению одновременно сегментного адреса и смещения.

Язык ассемблера поддерживает шесть следующих моделей памяти:

tiny (минимальная) – код и данные размещаются в одном сегменте размером до 64 Кбайт. И код и данные имеют тип near;

small (малая) – один сегмент размером до 64 Кбайт выделяется для кода и один сегмент для данных. И код и данные имеют тип near;

medium (средняя) – код программы может иметь размер более 64 Кбайт, а для данных выделяется один сегмент размером до 64 Кбайт. Код имеет тип far, а данные – near;

compact (компактная) – размер данных может превышать 64 Кбайт, а для кода выделяется один сегмент размером до 64 Кбайт. Код имеет тип near, а данные – far;

large (большая) – код и данные могут иметь размер более 64 Кбайт, и код и данные имеют тип far;

huge (максимальная) – код и данные могут иметь размер более 64 Кбайт, и код и данные имеют тип far.

В большинстве практических задач наиболее эффективной с точки зрения затрат памяти и быстродействия является модель small.

Директива .STACK позволяет задать размер стека, например

```
.STACK 100h
```

Здесь определен стек размером 100h (256) байт.

Директива .DATA задает начало сегмента данных программы. Соответствующий сегментный адрес будет автоматически занесен в предопределенный символ @data. Для получения доступа к данным в сегменте .DATA необходимо записать это значение в базовый сегментный регистр:

```
mov ax, @data
mov ds, ax
```

Так же можно загрузить этим значением и регистр es.

Директива .CODE задает начало сегмента кода.

Директива .FARDATA задает начало сегмента данных программы типа far. Сегментный адрес будет автоматически занесен в предопределенный символ @fardata.

Директива обеспечивает группировку сегментов программы в соответствии с соглашениями, принятыми фирмой Microsoft.
Если директива. DOSSEG не используется, то порядок расположения сегментов

4.6. Стандартные директивы определения сегментов

Программа на языке ассемблера может состоять из отдельных программных модулей, содержащихся в различных файлах. Каждый модуль, в свою очередь, состоит из инструкций процессора или директив ассемблера и заканчивается директивой END. Метка, стоящая после кода псевдооперации END, определяет адрес, с которого должно начаться выполнение программы и называется точкой входа в программу.

Каждый модуль также разбивается на отдельные части директивами сегментации, определяющими начало и конец сегмента. Наряду с простейшими директивами определения сегментов можно применять стандартные директивы SEGMENT ENDS и ASSUME.

Директива SEGMENT определяет начало любого сегмента, а директива ENDS его конец. В начале директив ставится имя сегмента.

В общем виде директивы записываются следующим образом:

```
seg_name SEGMENT align combine use ‘class’
……
Seg_name ENDS
```

Все параметры, записанные после слова SEGMENT, являются необязательными.

Параметр align (выравнивание) определяет границу памяти для стартового адреса сегмента. Здесь можно записать одно из следующих значений:

BYTE – разрешено использовать любой стартовый адрес;

WORD – стартовый адрес должен находиться на границе слова;

DWORD – стартовый адрес должен находиться на границе двойного слова;

PARA – стартовый адрес должен находиться на границе параграфа (16 байт);

PAGE – стартовый адрес должен находиться на границе страницы (256 байт);

Параметр combine (объединение) показывает, как должны объединяться одноименные сегменты из разных модулей. Здесь можно записать одно из следующих значений:

AT – сегмент размещается по определенному адресу в памяти, например, если записано

specseg SEGMENT AT 0FE00h

то стартовый адрес сегмента должен быть 0FE00h;

COMMON – сегменты могут перекрывать друг друга, размер объединенного сегмента равен размеру наибольшего из объединяемых сегментов;

PRIVATE – сегменты не будут объединяться с другими сегментами;

PUBLIC – сегменты объединяются в один сегмент, размер объединенного сегмента равен сумме размеров объединяемых сегментов;

STACK – сегменты объединяются в один, в построенном EXE-файле регистр SS адресует начало объединенного сегмента, а регистр SP – байт за последним словом объединенного сегмента.

Параметр use (использование) предназначен для процессоров, начиная с 80386.

Параметр class (класс) используется компоновщиком для определения последовательности расположения сегментов в памяти.

Каждый сегмент может быть также разбит на части. В общем случае информационные сегменты SS, ES и DS состоят из определений данных, а программный сегмент CS – из инструкций и директив, группирующих инструкции в блоки. Программный сегмент может разбиваться на части директивами определения процедур – некоторых выделенных блоков программы. Как и для определения сегмента, имеются две директивы определения процедуры (подпрограммы) – директива начала PROC и директива конца ENDP. Процедура имеет имя, которое должно включаться в обе директивы. В сегменте процедуры могут располагаться последовательно одна за другой или могут быть вложенными одна в другую.

Директива ASSUME сообщает ассемблеру о соответствии между сегментными регистрами и сегментами программы. Директива имеет следующий формат:

```
ASSUME <пара>[[, <пара>]]
ASSUME NOTHING
где <пара> -   это  <сегментный регистр>     :<имя сегмента>
               либо <сегментный регистр>     :NOTHING
```

Например, директива

```
ASSUME ES:A, DS:B, CS:C
```

сообщает ассемблеру, что для сегментирования адресов из сегмента A выбирается регистр ES, для адресов из сегмента B – регистр DS, а для адресов из сегмента C – регистр CS.

Таким образом, директива ASSUME дает право не указывать в командах (по крайней мере, в большинстве из них) префиксы – опущенные префиксы будет самостоятельно восстанавливать ассемблер.

В качестве особенностей директивы прежде всего следует отметить, что директива ASSUME не загружает в сегментные регистры начальные адреса сегментов, а только сообщает компилятору, что в программе будет сделана такая загрузка. Директиву *ASSUME* можно размешать в любом месте программы, но обычно ее указывают в начале сегмента команд, так как информация из нее нужна только при трансляции инструкций. При этом в директиве обязательно должно быть указано соответствие между регистром CS и данным сегментом кода, 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. каждому сегментному регистру, можно поставить в соответствие только один сегмент. В то же время на один и тот же сегмент могут указывать разные сегментные регистры. 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.

Так как директива ASSUME лишь сообщает о том, что все имена из таких-то программных сегментов должны сегментироваться по таким-то сегментным регистрам (в начале выполнения программы в этих регистрах ничего нет), то выполнение программы необходимо начинать с команд, которые загружают в сегментные регистры адреса соответствующих сегментов памяти.

Загрузка производится следующим образом. Пусть регистр DS необходимо установить на начало сегмента B. Для загрузки регистра необходимо выполнить присваивание вида 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 (см. ниже). Поэтому такую пересылку следует делать через другой, несегментный регистр, например, через AX:

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

Аналогичным образом загружается и регистр ES.

Регистра CS загружать нет необходимости, так как к началу выполнения программы этот регистр уже будет указывать, на начало сегмента кода. Такую загрузку выполняет операционная система, прежде чем передает управление программе.

Загрузить регистр SS можно двояко. Во-первых, его можно загрузить в самой программе так же, как DS или 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
...
S     ENDS
```

В таком случае загрузка S в регистр SS будет выполнена автоматически до начала выполнения программы.

4.7. Директива INCLUDE

Встречая директиву INCLUDE, ассемблер весь текст, хранящийся в указанном файле, подставит в программу вместо этой директивы. В общем случае обращение к директиве INCLUDE (включить) имеет следующий вид:

```
INCLUDE   <имя файла>
```

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.

Директива полезна, когда в разных программах используется один и тот же фрагмент текста; чтобы не выписывать этот фрагмент в каждой программе заново, его записывают в какой-то файл, а затем подключают к программам с помощью данной директивы.

4.8. Ссылки вперед

Хотя ассемблер и допускает ссылки вперед (т.е. к еще необъявленным объектам программы), такие ссылки могут при неправильном использовании приводить к ошибкам. Пример ссылки вперед:

```
JMP  MET
 ...
MET: ...
```

Всякий раз, когда ассемблер обнаруживает неопределенное имя на 1-м проходе, он предполагает, что это ссылка вперед. Если указано только имя, ассемблер делает предположения о его типе и используемом регистре сегмента, в соответствии с которыми и генерируется код. В приведенном выше примере предполагается, что MET – метка типа NEAR и для ее адресации используется регистр CS, в результате чего генерируется инструкция JMP, занимающая 3 байта. Если бы, скажем, в действительности тип ссылки оказался FAR, ассемблеру нужно было бы генерировать 5-байтовую инструкцию, что уже невозможно, и формировалось бы сообщение об ошибке. Во избежание подобных ситуаций рекомендуется придерживаться следующих правил:

1) Если ссылка вперед является переменной, вычисляемой относительно регистров ES, SS или CS, следует использовать оператор переключения сегмента. Если он не использован, делается попытка вычисления адреса относительно регистра DS.

2) Если ссылка вперед является меткой инструкции в команде JMP и отстоит не далее, чем на 128 байтов, можно использовать оператор SHORT. Если этого не делать, метке будет присвоен тип FAR, что не вызовет ошибки, но на 2-м проходе ассемблер для сокращения размера содержащей ссылку инструкции вынужден будет вставить дополнительную и ненужную инструкцию NOP.

3) Если ссылка вперед является меткой инструкции в командах CALL или JMP, следует использовать оператор PTR для определения типа метки. Иначе ассемблер устанавливает тип NEAR, и, если в действительности тип метки окажется FAR, будет выдано сообщение об ошибке.

4.9. Структура EXE- и COM- программы

Следует отметить, что какой-либо фиксированной структуры программы на языке Ассемблера нет, но для небольших EXE-программ с трехсегментной структурой типична следующая структура:

```
;Определение сегмента стека
STAK      SEGMENT STACK
          DB   256 DUP (?)
STAK      ENDS
;Определение сегмента данных
DATA      SEGMENT
SYMB      DB '#'    ;Описание переменной с именем SYMB
                    ;типа Byte и со значением «#»
. . .               ;Определение других переменных
DATA      ENDS
;Определение сегмента кода
CODE      SEGMENT
ASSUME    CS:CODE,DS:DATA,SS:STAK
;Определение подпрограммы
PROC1     PROC
     . . .          ;Текст подпрограммы
PROC1     ENDP
START:              ;Точка входа в программу START
     XOR  AX,AX
     MOV  BX,data   ;Обязательная инициализация
     MOV  DS,BX     ;регистра DS в начале программы
     CALL PROC1     ;Пример вызова подпрограммы
     . . .          ;Текст программы
     MOV  AH,4CH    ;Операторы завершения программы
     INT  21H
CODE      ENDS
END  START
```

В общем случае, взаимное расположение сегментов программы может быть любым, но чтобы сократить в командах число ссылок вперед и избежать проблем с префиксами для них, рекомендуется сегмент команд размешать в конце текста программы.

Сегмент стека в приведенной структуре описан с параметром STACK, поэтому в самой программе нет необходимости загружать сегментный регистр SS. Сегментный регистр CS тоже нет необходимости загружать, как уже отмечалось ранее. В связи с этим в начале программы 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.

Необходимо также заметить, что все предложения, по которым ассемблер заносит что-либо в формируемую программу (инструкции, директивы определения данных и т.д.) обязательно должны входить в какой-либо программный сегмент, размещать их вне программных сегментов нельзя. Исключение составляют директивы информационного характера, например, директивы EQU, директивы описания типов структур и записей. Кроме того, не рекомендуется размещать в сегменте данных инструкции, а в сегменте кода – описание переменных из-за возникающих в этом случае проблем с сегментированием.

Типичная структура COM-программы аналогична структуре EXE-программы, с той лишь разницей, что, как уже отмечалось выше, COM-программа содержит лишь один сегмент – сегмент кода, который включает в себя инструкции процессора, директивы и описания переменных.

```
;Определение сегмента кода
CODE SEGMENT
ASSUME    CS:CODE,DS:CODE,SS:CODE
ORG  100H      ;Начало необходимое для COM-программы
;Определение подпрограммы
PROC1     PROC
     . . .          ;Текст подпрограммы
PROC1     ENDP
START:
     . . .          ;Текст программы
     MOV  AH,4CH    ;Операторы завершения программы
     INT  21H
;===== Data =====
BUF  DB   6         ;Определение переменной типа Byte
. . .               ;Определение других переменных
CODE ENDS
END  START
```

5. Команды пересылки

5.1. Команда MOV

Команда MOV – основная команда пересылки данных, которая пересылает один байт или слово данных из памяти в регистр, из регистра в память или из регистра в регистр. Команда MOV может также занести число (непосредственный операнд) в регистр или память. В действительности команда MOV это целое семейство машинных команд микропроцессора. На приведенном ниже рисунке представлены различные способы, которыми в микропроцессоре можно пересылать данные из одного места в другое. Каждый прямоугольник означает здесь регистр или ячейку памяти. Стрелки показывают пути пересылки данных, которые

допускает микропроцессор. Необходимо также помнить, что все команды микропроцессора могут указывать только один операнд памяти.

Из рисунка видно, что запрещены пересылки из одной ячейки памяти в другую, из одного сегментного регистра в другой, запись непосредственного операнда в память. Это обусловлено тем, что в персональном компьютере отсутствуют соответствующие машинные команды. Если по алгоритму необходимо произвести одно из таких действий, то оно обычно реализуется в две команды, пересылкой через какой-нибудь несегментный регистр. Кроме того, командой MOV нельзя менять содержимое сегментного регистра CS. Это связано с тем, что регистровая пара CS:IP определяет адрес следующей выполняемой команды, поэтому изменение любого из этих регистров есть ничто иное, как операция перехода. Команда же MOV не реализует переход.

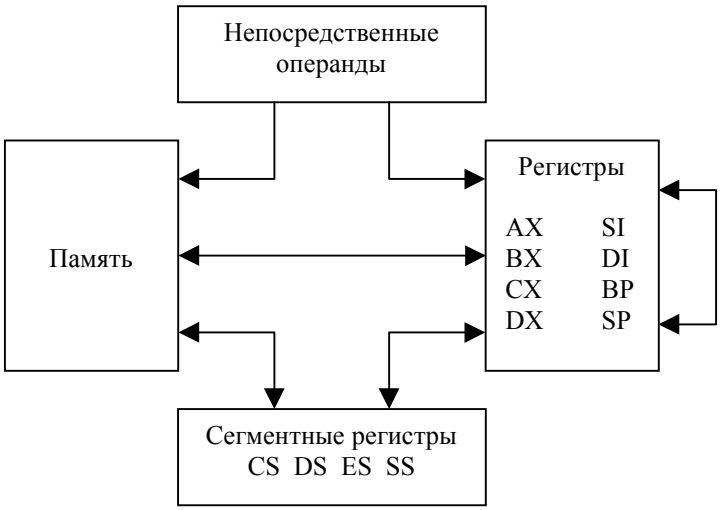

Рис. 5.1. Операнды команды пересылки MOV

Примеры использования команды пересылки:

```
MOV  Data,DI
MOV  BX,CX
MOV  DI,Index
MOV  Start_Seg,DS
MOV  ES,Buffer
MOV  Days,356
MOV  DI,0
```

5.2. Команда обмена данных XCHG

Команда XCHG меняет местами содержимое двух операндов. Порядок следования операндов не имеет значения. В качестве операндов могут выступать регистры (кроме сегментных) и ячейки памяти.

Примеры использования команды XCHG:

```
XCHG BL,BH
```

```
XCHG DH,Char
XCHG AX,BX
```

5.3. Команды загрузки полного указателя LDS и LES

Эти команды загружают полный указатель из памяти и записывают его в выбранную пару «сегментный регистр : регистр». При этом первое слово из адресуемой памяти загружается в регистр первого операнда, второе в регистр DS, если выполняется команда LDS, или в регистр ES если выполняется команда LES.

Примеры использования команд:

```
LDS  BX,[BP+4]
LES  DI,TablePtr
```

5.4. Команда перекодировки XLAT

Команда XLAT заменяет содержимое регистра AL байтом из таблицы перекодировки (максимальная длинна – 256 байт), начальный адрес которой относительно сегмента DS находится в регистре BX.

Алгоритм выполнения команды XLAT состоит из двух этапов:

– содержимое регистра AL прибавляется к содержимому регистра BX;

– полученный результат рассматривается как смещение относительно регистра DS. По данному адресу выбирается байт и помещается в регистр AL.

XLAT всегда использует в качестве смещения начала таблицы содержимое регистра BX, поэтому перед выполнением команды необходимо поместить в BX смещение таблицы.

Пример использования команды XLAT:

```
MOV       BX,OFFSET Talbe
MOV       AL,2
XLAT
...
Table     DB   'abcde'
```

5.5. Команды работы со стеком

Как уже было указано ранее, процессор адресует стек с помощью регистровой пары SS:SP. Помещение объектов в стек приводит к автоматическому декременту указателя стека, а извлечение – к инкременту, т.е. он «растет» в сторону меньших адресов памяти.

Для сохранения и восстановления различных 16-битовых данных в стеке используются команды PUSH (протолкнуть) и POP (вытолкнуть). За кодами операций PUSH и POP следует операнд, который необходимо поместить (извлечь) в (из) стек. В качестве операнда может выступать регистр или ячейка памяти, которую можно адресовать, используя известные способы адресации.

Замечание: Команда POP CS недопустима (восстановление из стека в регистр CS осуществляется по команде RET).

Для помещения в стек и извлечения из стека регистра флагов используются специальные команды PUSHF и POPF соответственно.

Стек удобен для передачи информации в подпрограммы и из них. Для этого подпрограмма может использовать BP как указатель на область стека. Ниже приведен фрагмент программы, демонстрирующий использование BP для доступа к параметрам, переданным через стек.

```
CODE SEGMENT
     ...
PROC1     PROC
     MOV  BP,SP     ;загрузка в BP текущего адреса стека
     MOV  BX,[BP+4];выборка из стека 1 параметра (ca)
     ...
     MOV  BX,[BP+2];выборка из стека 2 параметра (ll)
     ...
     RET  4         ;Возврат с удалением 4 слов из стека
PROC1     ENDP
START:
     ...
     MOV  AX,'ca'   ;Загрузка в AX символов
     MOV  CX,'ll'   ;Загрузка в CX символов
     PUSH AX        ;Сохранение AX в стек
     PUSH CX        ;Сохранение CX в стек
     CALL PROC1
     ...
CODE ENDS
```

5.6. Команды ввода-вывода

Все устройства ЭВМ принято делить на внутренние (центральный процессор ЦП, оперативная память ОП) и внешние (внешняя память, клавиатура, дисплей и т. д.). Под вводом-выводом понимается обмен информацией между ЦП и любым внешним устройством. В ЭВМ передача информации между ЦП и внешним устройством, как правило, осуществляется через порты. Порт – некоторый регистр размером в байт, находящийся вне ЦП (два соседних порта могут рассматриваться как порт размером в слово). Обращение к портам происходит по номерам. Все

порты нумеруются от 0 до 0FFFFh. С каждым внешним устройством связан свой порт или несколько портов, их адреса заранее известны.

Запись и чтение порта осуществляется при помощи следующих команд:

```
Чтение (ввод): IN   AL, n     или  IN   AX, n
Запись (вывод):OUT  n, AL     или  OUT  n, AX
```

Номер порта n в этих командах может быть задан либо непосредственно, либо регистром DX.

```
 IN  AX,DX
```

Сценарий ввода вывода через порты существенно зависит от специфики того внешнего устройства, с которым ведется обмен, но обычно ЦП связан с внешним устройством через два порта: первый – порт данных, второй – порт управления и достаточно типичной является следующая процедура обмена:

– ЦП записывает в порт управления соответствующую команду, а порт данных – выводимые данные;

– внешнее устройство, считав эту информацию, записывает в порт управления команду «занято» и начинает непосредственно вывод (например, печать);

– ЦП переходит либо в режим ожидания, опрашивая в цикле порт управления, либо занимается другой работой – до тех пор пока в порте управления не сменится сигнал «занято»;

– внешнее устройство заканчивает вывод и записывает в порт управления сигнал об успешном завершении или об ошибке;

– ЦП анализирует полученную информацию и продолжает свою работу.

6. Арифметические команды

Все арифметические команды устанавливают флаги CF, AF, SF, ZF, OF и PF в зависимости от результат операции.

Двоичные числа могут иметь длину 8 и 16 бит. Значение старшего (самого левого бита) задает знак числа: 0 – положительное, 1 – отрицательное. Отрицательные числа представляются в так называемом дополнительном коде, в котором для получения отрицательного числа необходимо инвертировать все биты положительного числа и прибавить к нему 1. Пример:

```
Положительное:        24=18h=       00011000b
Инверсное:                          11100111b
Отрицательное:                      11101000b    =E8h=-24
Проверка:             24-24=0       00011000b
                                    11101000b
                                   ----------
                                   (1)00000000b
```

6.1. Команды арифметического сложения ADD и ADC

Команда ADD выполнят целочисленное сложение двух операндов, представленных в двоичном коде. Результат помещается на место первого операнда, второй операнд не изменяется. Команда корректирует регистр флагов в соответствии с результатом сложения. Существуют две формы сложения: 8-битовое и 16-битовое. В различных формах сложения принимают участие различные регистры. Компилятор следит за тем, чтобы операнды соответствовали друг другу. На следующих рисунках иллюстрируются различные варианты команды ADD.

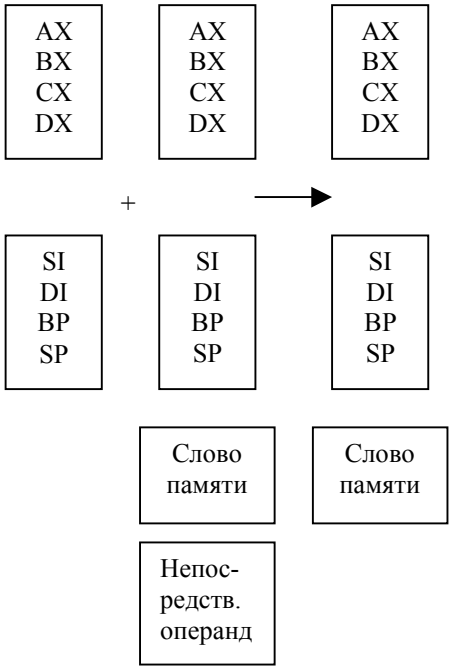

Рис. 6.1. 16-битовое сложение

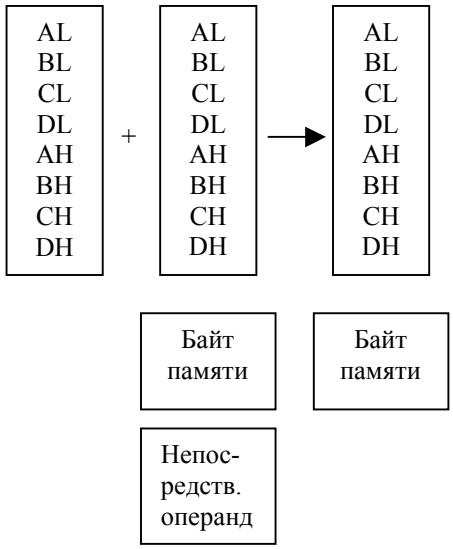

Рис. 6.2. 8-битовое сложение.

Команда сложения с переносом ADC – это та же команда ADD, за исключением того, что в сумму включается флаг переноса CF, который прибавляется к младшему биту результата. Для любой формы команды ADD существует аналогичная ей команда ADC. Команда ADC часто выполняется как часть многобайтной или многословной операции сложения.

Примеры использования команд ADD и ADC:

```
ADD  AL,12h
ADD  Count,1
ADC  BX,4
ADD  AX,BX
ADC  Count,DI
```

6.2. Команды арифметического вычитания SUB и SBB

Команда вычитания SUB – идентична команде сложения, за исключением того, что она выполняет вычитание, а не сложение. Для нее верны предыдущие схемы, если в них поменять знак «+» на «–», т. е. она из первого операнда вычитает второй и помещает результат на место первого операнда. Команда вычитания также устанавливает флаги состояния в соответствии с результатом операции (флаг переноса здесь трактуется как заем). Команда вычитания с заемом SBB учитывает флаг заема CF, то есть значение заема вычитается из младшего бита результата.

Примеры использования команд SUB и SBB:

```
SUB  AL,12h
SUB  Count,1
SBB  BX,4
SUB  AX,BX
SBB  Count,DI
```

6.3. Команда смены знака NEG

Команда отрицания NEG – оператор смены знака. Она меняет знак двоичного кода операнда – байта или слова.

6.4. Команды инкремента INC и декремента DEC

Команды инкремента и декремента изменяют значение своего единственного операнда на единицу. Команда INC прибавляет 1 к операнду, а команда DEC вычитает 1 из операнда. Обе команды могут работать с байтами или со словами. На флаги команды влияния не оказывают.

6.5. Команды умножения MUL и IMUL

Существуют две формы команды умножения. По команде MUL умножаются два целых числа без знака, при этом результат тоже не имеет знака. По команде IMUL умножаются целые числа со знаком. Обе команды работают с байтами и со словами, но для этих команд диапазон форм представления гораздо уже, чем для команд сложения и вычитания. На приведенных ниже рисунках представлены все варианты команд умножения.

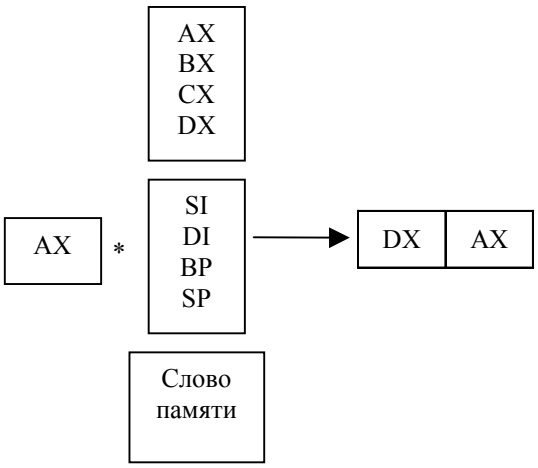

Рис. 6.3. Умножение слов.

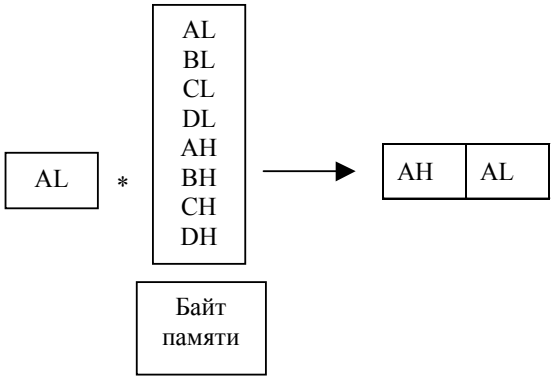

Рис. 6.4. Умножение байт.

При умножении 8-битовых операндов результат всегда помещается в регистр AX. При умножении 16-битовых данных результат, который может быть длиною до 32 бит, помещается в пару регистров: в регистре DX содержатся старшие 16-бит, а в регистре AX – младшие 16-бит. Умножение не допускает непосредственного операнда.

Установка флагов командой умножения отличается от других арифметических команд. Единственно имеющие смысл флаги – это флаг переноса и переполнения.

Команда MUL устанавливает оба флага, если старшая половина результата не нулевая. Если умножаются два байта, установка флагов переполнения и переноса показывает, что результат умножения больше 255 и не может содержаться в одном байте. В случае умножения слов флаги устанавливаются, если результат больше 65535.

Команда IMUL устанавливает флаги по тому же принципу, т. е. если произведение не может быть представлено в младшей половине результата, но только в том случае если старшая часть результата не является расширением знака младшей. Это означает, что если результат положителен, флаг устанавливается как в случае команды MUL. Если результат отрицателен, то флаги устанавливаются в случае, если не все биты кроме старшего, равны 1. Например, умножение байт с отрицательным результатом устанавливает флаги, если результат меньше 128.

Примеры использования команд умножения:

```
MUL  CX
IMUL Width
```

6.6. Команды деления DIV и IDIV

Как и в случае умножения, существуют две формы деления – одна для двоичных чисел без знака DIV, а вторая для чисел со знаком – IDIV. Любая форма деления может работать с байтами и словами. Один из операндов (делимое) всегда в два раза длиннее обычного операнда. Ниже приведены схемы, иллюстрирующие команды деления.

Байтовая команда делит 16-битовое делимое на 8-битовый делитель. В результате деления получается два числа: частное помещается в регистр AL, а остаток – в AH. Команда, работающая со словами, делит 32-битовое делимое на 16-битовый делитель. Делимое находится в паре регистров DX:AX, причем регистр DX содержит старшую значимую часть, а регистр AX – младшую. Команда деления помещает частное в регистр AX, а остаток в DX.

Ни один из флагов состояния не определен после команды деления. Однако, если частное больше того, что может быть помещено в регистр результата (255 для байтового деления и 65535 для деления слов), возникает ошибка значимости и выполняется программное прерывание уровня 0.

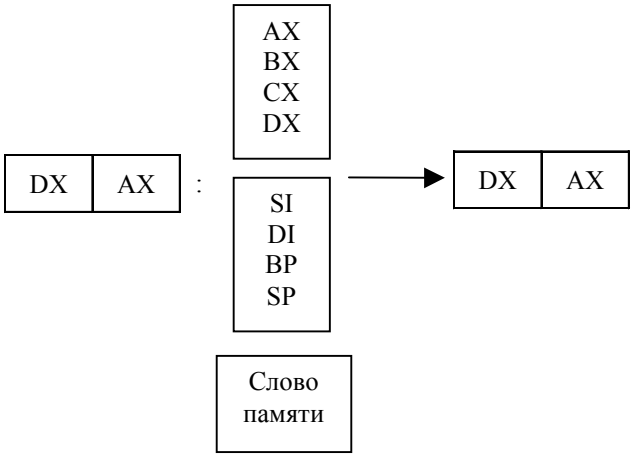

Рис. 6.5. Деление слов

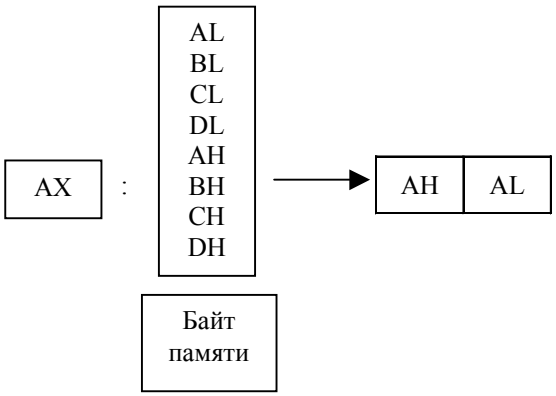

Рис. 6.6. Деление байт.

Примеры использования команд деления:

```
IDIV CX
DIV  Count
```

Рассмотрим пример программы, использующей большинство из описанных выше команд.

Пример. Вычислить значение арифметического выражения. Все числа являются 16-битовыми целыми со знаком. Формула вычислений следующая:

$$X = 1 - \frac{A*2 + B*C}{D - 3}$$

Эту задачу решает приведенная ниже программа.

```
;Сегмент стека
SSEG      SEGMENT   STACK
          DB   256  DUP(?)
SSEG      ENDS
;Сегмент данных
DATA      SEGMENT
     X    DW   ?          ;Память для переменных
     A    DW   ?
     B    DW   ?
     C    DW   ?
     D    DW   ?
DATA      ENDS
;Сегмент кода
CODE      SEGMENT
ASSUME    CS:CODE, DS:DATA, SS:SSEG
START:
     MOV       AX,Data    ;Инициализация DS
     MOV       DS,AX
;Вычислительная часть
     MOV       AX,2       ;Загрузка константы
     IMUL      A          ;dx:ax = a*2
     MOV       BX,DX
     MOV       CX,AX      ;bx:cx = a*2
     MOV       AX,B
```

```
        IMUL    C           ;dx:ax = b*c
        ADD     AX,CX
        ADC     DX,BX       ;dx:ax = a*2+b*c
        MOV     CX,D
        SUB     CX,3        ;cx = d-3
        IDIV    CX          ;ax = (a*2+b*c)/(d-3)
        NEG     AX          ;ax = -ax
        INC     AX          ;ax = ax+1
        MOV     X,AX        ;Сохранение результата
        MOV     AH,4CH
        INT     21H
CODE    ENDS
END     START
```

На первом этапе программа выполняет два умножения. Так как результат умножения всегда помещается в пару регистров DX:AX, то в примере результат первого умножения переносится в пару регистров BX:CX перед выполнением второго умножения. Затем программа выполняет сложение числителя. Поскольку умножение дает 32-битовые результаты, в программе требуется сложение повышенной точности (с учетом флага переноса). После сложения результат остается в DX:AX (числитель). Знаменатель вычисляется в регистре CX, а затем на него делится числитель. Частное записывается в регистр AX, затем его знак меняется на обратный и к полученному значению прибавляется 1. На последнем этапе программа записывает результат из регистра AX в переменную X. Остаток игнорируется.

7. Команды побитовой обработки

Эту группу команд можно разделить на две подгруппы: логические операции и операции сдвигов. Команды AND, TEST, OR, XOR и команды сдвигов изменяют значения флагов CF, OF, PF, SF, ZF (значение флага AF становится неопределенным). Команды циклических сдвигов изменяют только флаги OF и CF.

7.1. Команды, выполняющие логические операции

К командам, выполняющим логические операции, относятся AND, OR и XOR. Указанные команды выполняют соответственно операции «логическое умножение» (конъюнкцию), «логическое сложение» (дизъюнкцию) и «исключающее или» для двух операндов и помещают результат на место первого операнда. К группе логических команд также относится команда TEST, которая производит те же действия, что и команда AND, но не изменяет своих операндов, а лишь устанавливает соответствующие флаги.

В качестве операндов логических команд могут выступать те же операнды, что и у команд сложения и вычитания.

Примеры использования логических команд:

```
AND  BL,100
TEST CX,DX
OR   DX,Mask
XOR  Flag,1000b
```

7.2. Команды, выполняющие операции сдвигов

Команды сдвига перемещает все биты в поле данных либо вправо, либо влево, работая либо с байтами, либо со словами. Каждая команда содержит два операнда: первый операнд – поле данных – может быть либо регистром, либо ячейкой памяти; второй операнд – счетчик сдвигов. Его значение может быть равным 1, или быть произвольным. В последнем случае это значение необходимо занести в регистр CL, который указывается в команде сдвига. Число в CL может быть в пределах 0-255, но его практически имеющие смысл значения лежат в пределах 0-16.

Общая черта всех команд сдвига – установка флага переноса. Бит, попадающий за пределы операнда, сохраняется во флаге переноса. Всего существует 8 команд сдвига: 4 команды обычного сдвига и 4 команды циклического сдвига. Команды циклического сдвига переносят появляющийся в конце операнда бит в другой конец, а в случае обычного сдвига этот бит пропадает. Значение, вдвигаемое в операнд, зависит от типа сдвига. При логическом сдвиге вдвигаемый бит всегда 0, арифметический сдвиг выбирает вдвигаемый бит таким образом, чтобы сохранить знак операнда. Команды циклического сдвига с переносом и без него отличаются трактовкой флага переноса. Первые рассматривают его как дополнительный 9-ый или 17-ый бит в операции сдвига, а вторые нет.

Ниже приведен перечень команд сдвига:

– команды логического сдвига вправо SHR и влево SHL;

– команды арифметического сдвига вправо SAR и влево SAL;

– команды циклического сдвига вправо ROR и влево ROL;

– команды циклического сдвига вправо RCR и влево RCL с переносом;

Действие команд сдвига иллюстрируют следующие рисунки.

SHL SHR

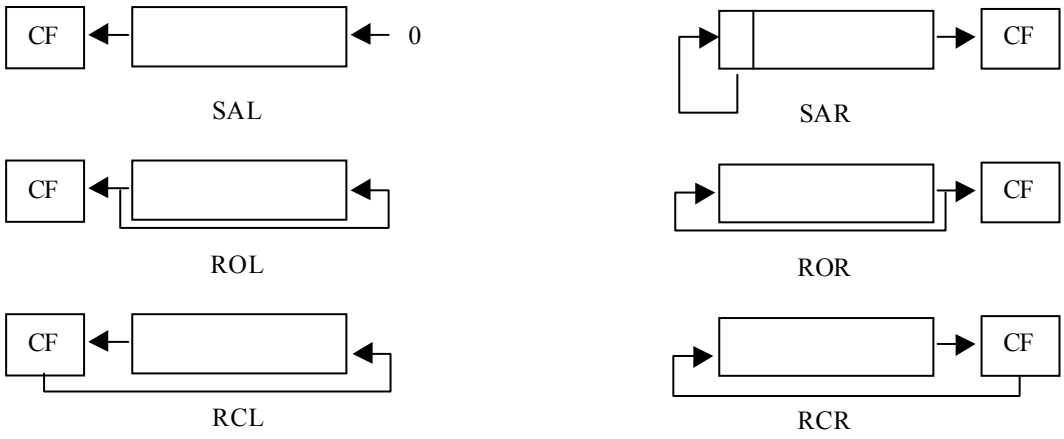

Рис. 7.1. Команды сдвига.

Примеры использования команд сдвига:

```
SHL  CH,1
SHL  [BP],CL
RCL  Size,1
```

Приведенная ниже программа иллюстрирует использование команд побитовой обработки.

Пример. Вывести на экран шестнадцатеричное представление кода символа «Q».

```
;Сегмент стека
SSEG      SEGMENT   STACK
     DB   256 DUP (?)
SSEG      ENDS
;Сегмент данных
DSEG      SEGMENT
SMP  DB   'Q'                  ;Символ
TBL  DB   '0123456789ABCDEF'  ;Таблица 16-ричных цифр
DSEG      ENDS
;Сегмент кода
CSEG      SEGMENT
ASSUME    CS:CSEG, DS:DSEG, SS:SSEG
START:
     MOV  AX,DSEG          ;Инициализация DS
     MOV  DS,AX
     MOV  AH,2             ;В AH номер функции вывода
     MOV  BX,0
;Вывод на экран цифры соответствующей левой тетраде
     MOV  BL,Smp           ;В BL символ
     MOV  CL,4             ;В CL величина сдвига
     SHR  BL,CL            ;Сдвиг левой тетрады на место правой
     MOV  DL,Tbl[BX]       ;Загрузка цифры  из таблицы в DL
     INT  21H              ;Вывод на экран
;Вывод на экран цифры соответствующей правой  тетраде
     MOV  BL,Smp           ;В BL символ
     AND  BL,00001111B     ;Обнуление левой тетрады
     MOV  DL,Tbl[BX]       ;Загрузка цифры  из таблицы в DL
```

```
     INT  21H            ;Вывод на экран
;Вывод на экран символа «h»
     MOV  DL,'h'
     INT  21H
CSEG      ENDS
END       START
```

Приведенная программы выводит на экран шестнадцатеричный кода буквы «Q», то есть выводятся две шестнадцатеричные цифры, соответствующие тетрадам байта, представляющего код символа «Q». Содержимое каждой тетрады как четырехбитового поля используется в качестве значения индекса для таблицы байтов, каждый из которых представляет шестнадцатеричный символ. Такой прием является распространенным способом перевода внутреннего битового представления во внешнее символьное.

8. Команды сравнения и передачи управления

Команда сравнения CMP сравнивает два числа, вычитая второе из первого, также как и команда SUB. Отличие команд CMP и SUB состоит в том, что инструкция CMP не сохраняет результат, а лишь устанавливает в соответствии с результатом флаги состояния. Основное назначение команды CMP – это организация ветвлений (условных переходов) в ассемблерных программах

Безусловный переход – это переход, который передает управление без сохранения информации возврата всякий раз, когда выполняется. Ему соответствует команда JMP. Эта команда может осуществлять переход вплоть до 32768 байт. Если заранее известно, что переход вперед делается на место, лежащее в диапазоне 128 байт от текущего места, можно использовать команду JMP SHORT LABEL. Атрибут SHORT заставляет Ассемблер сформировать короткую форму команды перехода, даже если он еще не встретил метку LABEL.

Условный переход проверяет текущее состояние машины (флагов или регистров), чтобы определить, передать управление или нет. Команды переходов по условию делятся на две группы:

– проверяющие результаты предыдущей арифметической или логической операции Jxx;

– управляющие итерациями фрагмента программы (организация циклов) LOOPxx.

Все условные переходы имеют однобайтовое смещение, то есть метка, на которую происходит переход должна находится в том же кодовом сегменте и на расстоянии, не превышающем –128 +127 байт от первого байта следующей команды. Если условный переход

осуществляется на место, находящееся дальше 128 байт, то вместо недопустимой команды

```
    JZ   ZERO
```

необходимо использовать специальные конструкции типа:

```
    JNZ  CONTINUE
    JMP  ZERO
CONTINUE:
```

Первая группа команд Jxx (кроме JCXZ/JECXZ) проверяет текущее состояние регистра флагов (не изменяя его) и в случае соблюдения условия осуществляет переход на смещение, указанное в качестве операнда. Флаги, проверяемые командой, кодируются в ее мнемонике, например: JC – переход, если установлен CF. Сокращения «L» (less – меньше) и «G» (greater – больше) применяются для целых со знаком, а «A» (above – над) и «B» (below – под) для целых без знака. Ниже в таблице показаны команды условного перехода и проверяемые ими флаги.

Таблица 8.1. Команды условного перехода.

Мнемоника	Флаги					Комментарии
	OF	CF	ZF	PF	SF	
Проверка флагов						
JE/JZ	X	X	1	X	X	
JP/JPE	X	X	X	1	X	
JO	1	X	X	X	X	
JS	X	X	X	X	1	
JNE/JNZ	X	X	0	X	X	
JNP/JPO	X	X	X	0	X	
JNO	0	X	X	X	X	
JNS	X	X	X	X	0	
Арифметика со знаком						
JL/JNGE	a	X	X	X	b	a не равно b (SF<>OF)
JLE/JNG	a	X	1	X	b	Z или a не равно b
JNL/JGE	a	X	X	X	b	a равно b
JNLE/JG	a	X	0	X	b	не Z и (a равно b)
Арифметика без знака						
JB/JNAE/JS	X	1	X	X	X	
JBE/JNA	X	1	1	X	X	CF или Z
JNB/JGE	X	0	X	X	X	
JNBE/JG	X	0	0	X	X	не CF или не Z

Буква X в любой позиции означает, что команда не проверяет флаг. Цифра 0 означает, что флаг должен быть сброшен, а цифра 1 означает, что флаг должен быть установлен, чтобы условие было выполнено (переход произошел).

Команды условного перехода можно разделить на три подгруппы:

1) Непосредственно проверяющие один из флагов на равенство 0 или 1.

2) Арифметические сравнения со знаком. Существуют 4 условия, которые могут быть проверены: меньше (JL), меньше или равно (JLE), больше (JG), больше или равно (JGE). Эти команды проверяют одновременно три флага: знака, переполнения и нуля.

3) Арифметические без знака. Здесь также существует 4 возможных соотношения между операндами. Учитываются только два флага. Флаг переноса показывает какое из двух чисел больше. Флаг нуля определяет равенство.

Ниже приведен фрагмент программы, иллюстрирующий использование команд сравнения и перехода.

```
CSEG SEGMENT
ASSUME CS:CSEG, DS:DSEG, SS:SSEG
START:
     ...
     MOV  BH,X      ;Загрузка в BH значения X
     MOV  BL,Y      ;Загрузка в BL значения Y
     CMP  BH,BL     ;Сравнение BH и BL
     JE   MET1      ;Если BH=BL, то переход на MET1
     JMP  MET2      ;Иначе переход на MET2
MET1:
     ...
     JMP  MET3
MET2:
     ...
MET3:
     MOV  AH,4Ch
     INT  21H
CSEG ENDS
END  START
```

JCXZ отличается от других команд условного перехода тем, что она проверяет содержимое регистра CX, а не флагов. Эту команду лучше всего применять в начале условного цикла, чтобы предотвратить вхождение в цикл, если CX=0.

Вторая группа команд условного перехода LOOPxx служит для организации циклов в программах. Все команды цикла используют регистр CX в качестве счетчика цикла. Простейшая из них – команда LOOP. Она уменьшает содержимое CX на 1 и передает управление на указанную метку, если содержимое CX не равно 0. Если вычитание 1 из CX привело к нулевому результату, выполняется команда, следующая за LOOP.

Команда LOOPNE (цикл пока не равно) осуществляет выход из цикла, если установлен флаг нуля или если регистр CX достиг нуля. Команда LOOPE (цикл пока равно) выполняет обратную описанной выше проверку флага нуля: в этом случае цикл завершается, если регистр CX достиг нуля или если не установлен флаг нуля.

Приведенный ниже фрагмент программы иллюстрирует использование команд организации циклов.

```
DSEG SEGMENT
BUF  DB   "0123406789"
DSEG ENDS
CSEG SEGMENT
     ASSUME    CS:CSEG,DS:DSEG,SS:SSEG
START:
     ...
     MOV  BX,OFFSET BUF;В BX – начало буферов
     MOV  CX,10        ;В CX – длинна буфера
     MOV  SI,0
M1:  MOV  DL,[BX+SI]   ;В DL – символ из буфера
     MOV  AH,2         ;в AH номер функции-вывода
     INT  21H          ;Вывод на экран
     INC  SI           ;Увеличение индекса на 1
     LOOP M1           ;Оператор первого цикла
     ...
CSEG ENDS
END  START
```

9. Подпрограммы и прерывания.

Все современные программы разрабатываются по модульному принципу – программа обычно состоит из одной или нескольких небольших частей, называемых подпрограммами или процедурами, и одной главной программы, которая вызывает эти процедуры на выполнение, передавая им управление процессором. После завершения работы процедуры возвращают управление главной программе и выполнение продолжается с команды, следующей за командой вызова подпрограммы.

Достоинством такого метода является возможность разработки программ значительно большего объема небольшими функционально законченными частями. Кроме того, эти подпрограммы можно использовать в других программах, не прибегая к переписыванию частей программного кода. В довершение ко всему, так как размер сегмента не может превышать 64К, то при разработке программ с объемом кода более 64К, просто не обойтись без модульного принципа.

Язык программирования Ассемблера поддерживает применение процедур двух типов – ближнего (near) и дальнего (far).

Процедуры ближнего типа должны находится в том же сегменте, что и вызывающая программа. Дальний тип процедуры означает, что к ней можно обращаться из любого другого кодового сегмента.

При вызове процедуры в стеке сохраняется адрес возврата в вызывающую программу:

– при вызове ближней процедуры – слово, содержащее смещение точки вызова относительно текущего кодового сегмента;

– при вызове дальней процедуры – слово, содержащее адрес сегмента, в котором расположена точка возврата, и слово, содержащее смещение точки возврата в этом сегменте.

В общем случае группу команд, образующих подпрограмму, можно никак не выделять в тексте программы. Для удобства восприятия в языке Ассемблера процедуры принято оформлять специальным образом. Описание процедуры имеет следующий синтаксис:

```
<имя_процедуры>     PROC <параметр>
                    <тело_процедуры>
<имя_процедуры>     ENDP
```

Следует обратить внимание, что в директиве PROC после имени не ставится двоеточие, хотя имя и считается меткой.

Параметр, указываемый после ключевого слова PROC, определяет тип процедуры: ближний (NEAR) или дальний (FAR). Если параметр отсутствует, то по умолчанию процедура считается ближней.

В общем случае, размещать подпрограмму в теле программы можно где угодно, но при этом следует помнить, что сама по себе подпрограмма выполняться не должна, а должна выполняться лишь при обращении к ней. Поэтому подпрограммы принято размещать либо в конце сегмента кода, после команд завершения программы, либо в самом начале сегмента кода, перед точкой входа в программу. В больших программах подпрограммы нередко размещают в отдельном кодовом сегменте.

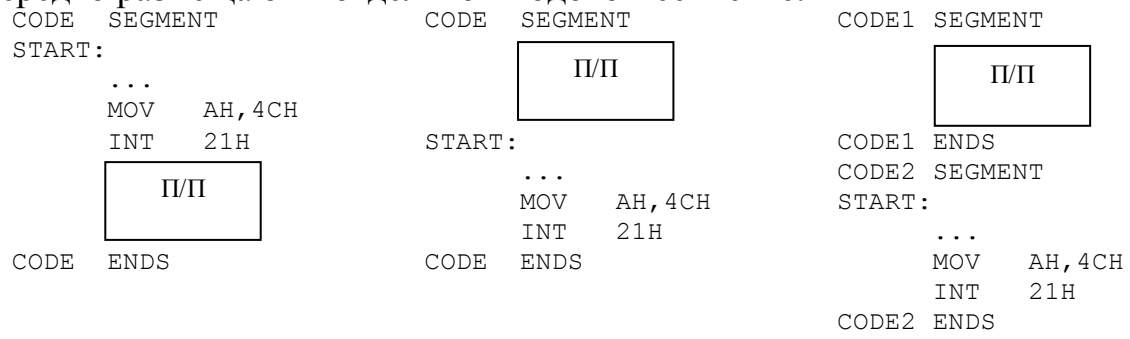

Рис. 9.1. Варианты размещения подпрограммы в теле программы.

Передавать фактические параметры процедуре можно несколькими способами. Простейший способ – передача параметров через регистры: основная программа записывает параметры в какие-либо регистры, а процедура по мере необходимости извлекает их из этих регистров и использует в своей работе. Такой способ имеет один основной недостаток: передавать параметры через регистры можно если их немного (если много, то просто не хватит регистров). Решить это проблему можно, передавая параметры через стек. В этом случае основная программа записывает параметры в стек и вызывает подпрограмму, подпрограмма работает с

параметрами и, возвращая управление, очищает стек (см. пример в разделе «5.5. Команды работы со стеком»).

Для работы с подпрограммами в систему команд процессора включены специальные команды, это вызов подпрограммы CALL и возврат управления RET.

Все команды вызова CALL безусловны. Внутрисегментный вызов NEAR CALL используется для передачи управления процедуре, находящейся в том же сегменте. Он указывает новое значение регистра IP и сохраняет старое значение счетчика команд (IP) в стеке в качестве адреса возврата. Межсегментный вызов FAR CALL используется для передачи управления процедуре, находящейся в другом сегменте или даже программном модуле. Он задает новые значения сегмента CS и смещения IP для дальнейшего выполнения программы и сохраняет в стеке как регистр IP, так и регистр CS.

Все возвраты RET являются косвенными переходами, поскольку извлекают адрес перехода из вершины стека. Внутрисегментный возврат извлекает из стека одно слово и помещает его в регистр IP, а межсегментный возврат извлекает из стека два слова, помещая слова из меньшего адреса в регистр IP, а слово из большего адреса – в регистр CS. Команда RET может иметь операнд, который представляет собой значение, прибавляемое микропроцессором к содержимому указателя стека SP после извлечения адреса возврата (очистка стека) (см. пример в «5.5. Команды работы со стеком».

Другим видом процедур являются прерывания DOS и BIOS. Прерывания это обычные процедуры, написанные разработчиками операционной системы (прерывания DOS) или разработчиками аппаратных средств компьютера (прерывания BIOS). Поэтому к этим процедурам можно обращаться точно так же, как и к другим процедурам. Отличает их лишь форма вызова: вместо команды CALL ProcName используется команда типа INT Number, где Number – номер прерывания, например INT 10h, INT 21h и т.п. Эта команда вызывает прерывание с номером Number (0< Number <255), то есть после такой команды начинает работать подпрограмма обработки прерывания, начальный адрес которой записан в Number-ом элементе таблицы векторов прерываний. (Таблица векторов прерываний представляет собой массив из 256 элементов, расположенный по адресу 0000:0000 и имеющий длину 1024 байт. Каждый элемент таблицы векторов занимает 4 байта и представляет собой начальный адрес процедуры обработки прерывания.) Закончив свое выполнения, подпрограмма передает управление на команду расположенную непосредственно за командой INT.

В связи с тем, что в состав операционной системы входит много стандартных процедур и для них не хватает допустимых векторов

прерываний, они (процедуры) объединяются в группы, вызываемые по прерыванию с одним и тем же номером. Подпрограммы одной группы называют функциями прерывания. Чтобы различать функции прерывания перед его вызовом в регистр AH заносится номер необходимой функции:

```
MOV  AH,<номер функции>
INT  <номер прерывания>
```

Для выполнения вызванной таким образом процедуры может потребоваться определенная информация (например, для функции вывода символа на экран – код символа). Такая информация передается через регистры микропроцессора.

Ниже приведен пример использования прерывания 21H функции 02 для вывода символа на экран.

```
...
MOV  AH,02h     ;AH – номер функции
MOV  DL,'a'     ;DL – выводимый символ
INT  21h        ;инициализация прерывания
...
```

10. Команды работы со строками

Строкой в Ассемблере называют последовательность байтов или слов длинной от 1 до 65535 байт. Операции со строками обеспечивают пересылку, сравнение, сканирование строк по значению, а также пересылку строки в аккумулятор или из него. Каждая строковая операция представленная в процессоре двумя командами: одна предназначена для обработки строк состоящих из байт, другая – из слов (их мнемоника различается наличием буквы B (byte) или W (word)).

Если флаг направления DF перед выполнением команды строковой обработки установлен в 0 (выполнена команда CLD), значение в индексном регистре автоматически увеличивается, если в 1 (выполнена STD) – уменьшается. Индексные регистры уменьшаются или увеличиваются на 1, если команды работают с байтами, или на 2 – если со словами.

Команды строковой обработки чаще всего используются с однобайтными префиксами (префиксами повторения), которые обеспечивают многократное автоматическое повторение выполнения команды.

Команда сравнения строк CMPS (CMPSB, CMPSW).

Команда CMPS сравнивает значение элемента одной строки (DS:SI) со значением элемента второй строки (ES:DI) и настраивает значения регистров на следующие элементы строк в соответствии с флагом направления DF. Сравнение происходит так же, как и по команде сравнения CMP. Результатом операции является установка флагов.

Команда сканирования строки SCAS (SCASB, SCASW).

Команда SCAS производит сравнение содержимого регистра (AL или AX) с байтом или словом памяти, абсолютный адрес которого определяется парой ES:DI, после чего регистр DI устанавливается на соседний элемент памяти (байт или слово) в соответствии с флагом DF. Команда SCAS используется обычно для поиска в строке (ES:DI) элемента заданного в регистре AL или AX.

Команда пересылки строки MOVS (MOVSB, MOVSW).

Команда MOVS пересылает поэлементно строку DS:SI в строку ES:DI и настраивает значения индексных регистров на следующий элемент строки.

Команда сохранения строки STOS (STOSB, STOSW).

Команда STOS заполняет строку, содержащуюся по адресу ES:DI, элементом из регистра AL или AX. На флаги команда не влияет

Команда загрузки строки LODS (LODSB, LODSW).

Команда LODS записывает в регистр AL или AX содержимое ячейки памяти, адрес которой задается регистрами DS:SI. Флаги не меняются.

Префиксы повторения.

В системе команд процессора имеются команды без операндов, которые называются префиксами повторения:

REPE (повторять, пока равно);
REPZ (повторять, пока ноль);
REP (повторять);
REPNE (повторять, пока не равно);
REPNZ (повторять, пока не ноль).

Префиксы повторения ставятся перед строковыми командами обязательно в той же строке, например:

```
REPE CMPB
```

Префикс использует регистр CX как счетчик циклов. На каждом этапе цикла выполняются следующие действия:

1) Проверка CX. Если он равен 0 – выход из цикла и переход к следующей команде.
2) Подтверждение любых возникающих прерываний.
3) Выполнение указанной строковой операции.
4) Уменьшение CX на единицу, флаги при этом не изменяются.
5) Проверка флага ZF, если выполняется строковая операция SCAS или CMPS. Если условие повторения цикла не выполняется – выход из цикла и переход к следующей команде. Выход из цикла, если префиксом является REPE и ZF=0 (последнее сравнение не совпало) или используется префикс REPNE и ZF=1 (последнее сравнение совпало).

6) Изменение значения индексных регистров в соответствии со значением флага направления и переход на начало цикла.

Фрагмент программы, иллюстрирующий работу со строковыми данными, приведен ниже.

```
     CLD             ;DF=0
     LEA  SI,s1      ;DS:SI=начало s1
     LEA  DI,s2      ;ES:DI=начало s2
     MOV  CX,n       ;CX=длинна строк
     REPE CMPSB      ;сравнение, пока элементы равны
     JNE  NoEq       ;если s1<>s2 (ZF=0), то на NoEq
     ...
NoEq:
     ...
```

11. Команды управления процессором

К командам управления процессором чаще всего относят команды работы (установка и очистка) с флагами. Среди них наиболее часто приходится использовать следующие.

Команда CLC устанавливает значение флага переноса CF, равное нулю. Все остальные флаги и регистры остаются неизменными.

Команда CMC изменяет значение флага переноса CF на противоположное. Другие флаги остаются без изменений.

Команда STC устанавливает флаг переноса в единицу.

Команда CLD очищает флаг направления DF. Все остальные флаги и регистры остаются неизменными. После выполнения CLD используемые строковые операции будут увеличивать индексный регистр (SI или DI).

Команда STD устанавливает флаг направления DF в единицу, что заставляет все последующие строковые операции уменьшать при их выполнении индексные регистр (SI или DI).

Команда CLI очищает флаг прерываний, в результате чего процессор не распознает внешние маскируемые прерывания.

Команда STI устанавливает флаг разрешения прерываний FI в единицу. После этого при завершении работы следующей команды процессор может выполнять обработку внешних прерываний, если эта команда снова не сбросит флаг прерываний.

12. Структуры данных

12.1. Массивы

Как уже было отмечено выше, для описания массивов в языке ассемблера используется директива DUP. Например, одномерный массив байт можно описать следующим образом:

```
Mas1 DB   10 DUP (?)
```

а двумерный массив слов:

```
Mas2 DW   10 DUP (10 DUP (?))
```

Легко заметить что, при описании массивов разработчик указывает имя массива, тип элементов и их количество, но не указывает как индексируются эти элементы. Поэтому, в общем случае, начальным индексом массива может быть любая величина k, например, для описанного выше одномерного массива, диапазон индексов будет следующим [k...9+k]. Тогда верно следующее соотношение:

```
Адрес(X[i])=X+(Type X)*[i-k]
```

где X – начальный адрес массива, (Type X) – тип элементов массива, k – индекс первого элемента массива. Очевидно, что при k=0 эта зависимость становится наиболее простой:

```
Адрес(X[i])=X+(Type X)*i
```

Таким образом, в случае, когда элементы массива нумеруются с нуля, доступ к ним значительно упрощается. Поэтому, в большинстве случаев именно так и делается.

Для многомерных массивов ситуация аналогична.

Теперь рассмотрим каким образом организуется доступ к элементам массива. Основная трудность возникают в ситуации, когда массив обрабатываются в цикле, и необходимо организовать доступ к каждому элементу массива. В этом случае поступают следующим образом:

– адрес текущего элемента массива X+(Type X)*i разбивается на два слагаемых: постоянное слагаемое X (начальный адрес массива) и переменное слагаемое (Type X)*i (смещение текущего элемента относительно начала массива);

– постоянное слагаемое записывается в саму команду, а переменное слагаемое заносится в какой-либо регистр модификатор, который также указывается в команде.

Таким образом, изменяя в цикле значение регистра-модификатора, можно на каждой итерации цикла получить доступ к новому элементу массива.

Ниже приведен фрагмент программы, иллюстрирующий нахождение суммы элементов одномерного массива байт, описанного выше.

```
MOV  AX,0      ;начальное значение суммы
MOV  CX,10     ;счетчик циклов (кол-во элементов)
```

```
     MOV  SI,0      ;начальное значение индекса
M:   ADD  AX,Mas1[SI]    ;AX:=AX+Mas1[i]
     INC  SI        ;увеличение индекса
     LOOP M
```

12.2. Связанные списки

Списком (линейным, однонаправленным) называется последовательность звеньев, которые могут размещаться в произвольных местах памяти и в каждом из которых содержится элемент списка (Ei) и ссылка на следующее звено (изображена стрелкой):

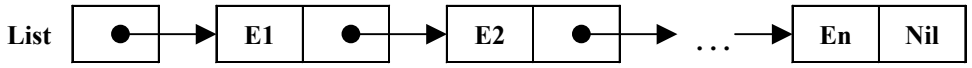

Рис 12.1. Структура связанного списка

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. 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 доступ к элементам списка осуществляется последовательно.

Для хранения звеньев списков обычно отводят специальную область памяти, называемую кучей (heap). Обычно 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. Ссылка на звено – это адрес данного звена. Для экономии памяти ссылкой на звено обычно считается лишь смещение A, то есть адрес этого звена, отсчитанный от начала кучи (два байта). Под каждое звено отводится несколько соседних байтов памяти, в первых из которых размещается соответствующий элемент списка ELEM, а в остальных ссылка на следующее звено NEXT. Размер звена зависит от размера элементов списка.

При программировании на языке ассемблера 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 необходимо получить доступ к полям звена, который имеет адрес A, то это осуществляется следующим образом:

```
ES:A.ELEM – поле с элементом
ES:A.NEXT – поле с адресом следующего звена
```

В качестве пустой ссылке nil обычно используется адрес 0, при этом удобно описать в программе этот адрес как константу

```
NIL  EQU  0
```

и далее пользоваться именем NIL.

Ссылки на первые звенья списков, как правило, хранятся в переменных из сегмента данных, имеют размер 16 бит и описываются, например, следующим образом:

```
List DW   ?
```

При работе со списком приходится последовательно просматривать его звенья. Поэтому необходимо знать адрес текущего звена. Для хранения этого адреса удобно использовать какой-нибудь регистр-модификатор, например, BX. Если текущим является i-е звено списка, то графически это можно представить так:

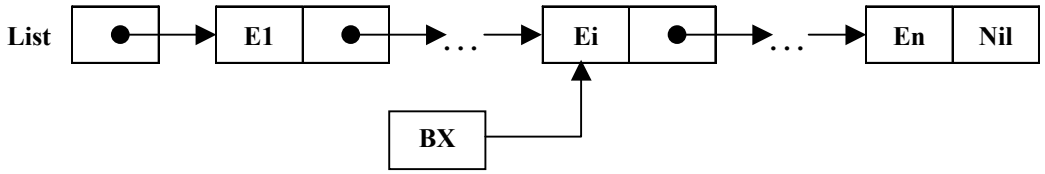

Рис 12.2. Указатель на текущее звено списка

В этом случае доступ к полям текущего звена можно получить следующим образом ES:[BX].ELEM и ES:[BX].NEXT.

Анализ текущего элемента списка

Следующий фрагмент программы иллюстрирует анализ, равенства элемента из текущего звена значению некоторой переменной X, и, если условие истинно, переход на метку EQ:

```
MOV       AX,ES:[BX].ELEM
CMP       AX,X
JE        EQ
```

Переход к следующему звену списка

Фрагмент программы, иллюстрирующий эту операцию, выглядит следующим образом (BX – текущее звено списка):

```
MOV       BX,ES:[BX].NEXT
```

Поиск элемента в списке

Нижеследующий фрагмент программы отыскивает в списке LIST значение X. Если X найден, то в регистр AL заносится 1 (входит) иначе 0.

```
     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
     CMP  ES:[BX].ELEM,CX
```

```
     JE   YES
     MOV  BX,ES:[BX].NEXT ;bx - адрес следующего звена
     JMP  L
YES: MOV  AL,1
NO:  ...
```

Вставка элемента в список

Для вставки нового элемента X в список LIST необходимо выполнить следующие действия (см. рис. 12.3):

1) отвести (выделить) место под новое звено (эти действия выполняет процедура NEW (см. ниже), она отыскивает в куче свободное место и возвращает его адрес через регистр DI);

2) заполнить новое звено, то есть в поле ELEM записать величину X, а в поле NEXT – ссылку на бывшее первое звено (она берется из LIST),

3) в LIST записать адрес нового звена (берется из DI):

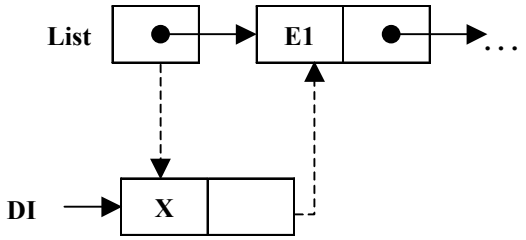

Рис 12.3. Вставка элемента в список

```
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
```

Организация кучи

При программировании на языке ассемблера необходимо самому освобождать и выделять память под звенья списка (процедуры New и Dispose). Ниже приведен один из вариантов организации кучи.

Список свободной памяти

В процессе работы программы память постоянно освобождается и выделяется, то есть занятые и свободные ячейки в сегменте кучи разбросаны. Поэтому удобно все свободные ячейки кучи объединить в один список, который принято называть списком свободной памяти (ССП). Начало этого списка указывается в некоторой фиксированной ячейке, например, с именем HEAP_PTR (heap pointer, указатель кучи). Используется ССП следующим образом: 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 в простейшем случае звенья имеют фиксированную длину, то имеет смысл объединять в ССП элементы именно этого размера (например, двойные слова, как в предыдущих примерах). Если при этом ссылку на следующее звено ССП хранить во втором слове звена, то получим обычный список.

Место для переменной HEAP_PTR лучше всего отвести в самом начале кучи – в ячейке с относительным адресом 0. В этом случае ни одно звено не будет размещаться в этой ячейке и не будет иметь адрес 0, т.е. этот адрес останется свободным и его можно использовать для представления пустой ссылки nil.

Описание сегмента кучи

Поскольку куча – это один из сегментов памяти, используемый программой, то его необходимо описать в программе (n – число звеньев в куче):

```
HEAP_SIZE EQU  N
HEAP SEGMENT             ;сегмент кучи
     HEAP_PTR  DW   ?    ;ячейка с начальным адресом ССП
     DD HEAP_SIZE DUP(?)
HEAP ENDS
```

Инициализация кучи

Первоначально все ячейки кучи свободны, и не объединены в ССП. Поэтому в начале программы необходимо связать все ячейки (двойные слова) кучи в единый список и записать в HEAP_PTR ссылку на начало этого списка. Нижи приведен пример подпрограммы инициализации кучи:

```
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
     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 необходимо 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, например, передается через регистр DI. Звено становится свободным и его необходимо добавить в начало ССП:

```
;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 ей необходимо место для нового звена списка. Это место берется из ССП: от этого списка отщепляется одно из звеньев и отдается программе. Проще всего отделить первое звено. Адрес этого звена процедура должна вернуть через регистр DI.

```
;на выходе: DI - адрес свободного звена
NEW  PROC FAR
     MOV  DI,ES:HEAP_PTR
     CMP  DI,NIL
     JE   EMPTY          ;пустой ССП -> EHPTY_HEAP
     PUSH ES:[DI].NEXT
     POP  ES:HEAP_PTR    ;heap_ptr - на 2-е звено ССП
     RET
     EMPTY:              ;реакция на пустую кучу
     ...
NEW  ENDP
```

Замечание: Приведенные варианты процедур NEW и DISPOSE рассчитаны только на случай, когда звенья всех списков имеют один и тот же размер.

13. Условное ассемблирование

Язык ассемблера включает в себя директивы условного ассемблирования. Условное ассемблирование – удобно при многократных прогонах программы. Оно дает возможность в исходном тексте держать несколько вариантов одного и того же участка программы, и при каждом 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. Таким образом, автор программы не должен перед каждым ее прогоном вручную редактировать текст, а возлагает эту работу на макрогенератор.

Участок программы, затрагиваемый условным ассемблированием, должен записываться в виде так называемого IF-блока:

```
<IF-директива>
<фрагмент-1>
ELSE
<фрагмент-2>
ENDIF
```

или

```
<IF-директива>
<фрагмент-1>
ENDIF
```

Директивы ELSE и ENDIF обязательно должны записываться в отдельных строках. 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-дирсктиве 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 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 (в нем не должно быть ссылок вперед).

В макроязыке довольно много IF-директив их удобно рассматривать парами, в каждой из которых директивы проверяют противоположные условия.

Директивы IF и IFE имеют следующий вид:

```
IF <константное выражение>
IFE <константное выражение>
```

Встречая любую из этих директив, макрогенератор вычисляет указанное в ней константное выражение. В директиве IF условие считается выполненным, если значение выражения отлично от 0, а директиве IFE - если значение равно 0.

В общем случае константное выражение, указываемое в директивах IF и IFE, может быть любым, но по смыслу оно, должно быть логическим выражением. В языке ассемблера 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: EQ -равно, NE - не равно, LT - меньше, LE -меньше или равно, GT - больше, GE - 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 если выполняется – «ложь».

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». Их операндами могут быть любые 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.

Замечание. Не следует путать операторы и команды: операторы используются для записи операндов команд и директив и вычисляются на этапе трансляции программы (в машинной программе их нет), а команды выполняются на этапе счета программы.

Директивы IFIDN, IFDIF, IFB и IFNB имеют следующий вид:

```
IFIDN <tl>,<t2>
IFDIF <tl>,<t2>
```

где tl и t2 - любые последовательности символов (причем они обязательно должны быть заключены в угловые скобки), которые посимвольно сравниваются. В директиве IFIDN условие считается выполненным, если эти строки равны, а в директиве IFDIF - если они не равны. При сравнении этих текстов большие и малые буквы не отождествляются.

Обе эти директивы имеет смысл использовать лишь внутри тела макроса, 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.

Директивы

```
IFB <t>
IFNB <t>
```

Фактически являются вариантами директив IFIDN и IFDIF, когда второй текст пуст. В директиве IFB условие считается выполненным, если текст t пуст, а в директиве IFNB - если текст t не пуст.

Эти директивы используются в макросах для проверки, задан ли фактический параметр или нет.

Ниже представлен пример использования директив условного ассемблирования.

Предположим, что необходимо отладить программу и для этого в определенные места ее текста вставляются отладочные печати (ОП), т. е. печать промежуточных значений каких-то переменных. После окончания отладки ОП убираются из текста. Процесс отладки может повторяться несколько раз. Вставлять и убирать ОП можно, с помощью какого-либо текстового редактора, но чаще всего ОП разбросаны по всей программе, и это занимает не мало времени. В подобной ситуации удобно использовать возможности условного ассемблирования: в тексте программы постоянно сохраняются ОП, но перед каждой из них указывается условие, что команды ОП должны оставаться в окончательном тексте программы только при отладке.

Это можно осуществить следующим образом. Пусть режим прогона программы указывается с помощью константы DEBUG, которая описана в начале текста программы и которой присваивается значение 1 (отладка) или значение 0 (счет). Тогда, (см. пример) участок исходной программы с отладочной печатью (например, переменной X) должен быть записан так, как указано слева, в окончательном же тексте программы этот участок будет выглядеть так, как изображено справа (PRINT – макрос вывода на экран):

```
…                          Debug<>0
MOV X,AX                        …
IF DEBUG                        MOV X,AX
PRINT X                         PRINT X
ENDIF                           MOV BX,0
MOV BX,0                        …
…                          Debug=0
                                MOV X,AX
                                MOV BX,0
                                …
```

При таком построении текста программы достаточно перед ее прогоном поменять лишь одну строчку – описание константы DEBUG, чтобы макрогенератор сформировал разный окончательный текст программы (с ОП или без них).

Включать и исключать ОП в текст программы можно и с помощью команд условного перехода. Основное отличие условного ассемблирования и команд условного перехода состоит в том, что в последнем случае команды ОП остаются в программе всегда, и потому

всегда занимают место в памяти. Кроме того, в этом случае 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, а также тратить время на проверку условия.

14. Макросредства

Макросредства языка ассемблера позволяют формировать в исходной программе блоки предложений (макроопределения), имеющие одно общее имя, и затем многократно использовать это имя для представления всего блока. В процессе ассемблирования компилятор автоматически замещает каждое распознаваемое макроимя (макрокоманду) последовательностью предложений в соответствии с макроопределением, в результате чего формируется макрорасширение.

Макрокоманда может встречаться в исходной программе столько раз, сколько это необходимо. Макроопределение в исходном файле должно предшествовать его первому использованию в макрокоманде. Макроопределение может как непосредственно находиться в тексте программы, так и подключаться из другого файла директивой INCLUDE. В макроопределение могут быть переданы параметры, которые будут управлять макроподстановкой или задавать фрагменты текста.

В программе на языке ассемблера макрокоманды выполняют в целом те же функции, что и процедуры, т.е. обеспечивают многократное и функционально законченное действие с параметрическим управлением. Различие заключается в следующем:

1) Процедура присутствует в исходной программе один раз, тогда как тело макроопределения дублируется столько раз, сколько соответствующих макрокоманд содержит исходный файл.

2) Текст процедуры статичен и неизменен в то время, как состав макрорасширения может зависеть от параметров макрокоманды.

Следует помнить, что параметры макрокоманды - это значения времени ассемблирования, а параметры процедуры принимают какие-то определенные значения лишь в процессе выполнения программы.

14.1. Макродирективы

Макроопределение представляет собой блок исходных предложений, начинающийся директивой MACRO и заканчивающийся директивой ENDM. Формат макроопределения:

```
имя MACRO [[формальный-параметр,...]]
```

```
предложения
ENDM
```

Именем макроопределения считается имя, указанное в директиве MACRO. Оно должно быть уникальным и может использоваться в исходном файле для вызова макроопределения. Формальные параметры представляют собой внутренние по отношению к данному макроопределению имена, которые используются для обозначения значений, передаваемых в макроопределение при его вызове. Может быть определено любое число формальных параметров, но все они должны помещаться на одной строке и разделяться запятыми.

В теле макроопределения допустимы любые предложения языка ассемблера, в том числе и другие макродирективы. Допустимо любое число предложений, а каждый формальный параметр может быть использован любое число раз в этих предложениях.

Макроопределения могут быть переопределены. При этом нет необходимости заботиться об удалении из исходного файла первого макроопределения, так как каждое следующее макроопределение автоматически замещает предыдущее макроопределение с тем же именем.

Макроопределение может быть вызвано в любой момент простым указанием его имени в исходном файле. Ассемблер при этом копирует предложения макроопределения в точку вызова, замещая в этих предложениях формальные параметры на фактические параметры, задаваемые при вызове.

Общий вид макровызова:

```
имя [[фактический-параметр,...]]
```

Имя должно быть именем ранее определенного в исходном файле макроопределения. Фактическим параметром может быть имя, число или другое значение. Допустимо любое число фактических параметров, но все они должны помещаться на одной строке. Элементы списка параметров должны разделяться запятыми, пробелами, или TAB-символами.

Компилятор замещает первый формальный параметр на первый фактический параметр, второй формальный параметр на второй фактический параметр и т.д. Если фактических параметров в макровызове больше, чем формальных параметров в макроопределении, лишние фактические параметры игнорируются. Если же фактических параметров меньше, чем формальных, формальные параметры, для которых не заданы фактические, замещаются пустыми строками (пробелами). Для определения того, заданы или не заданы соответствующие фактические параметры могут быть использованы директивы IFB и IFNB (см. «13. Условное ассемблирование»).

В макросредствах языка ассемблера имеется возможность передавать список значений в качестве одного параметра. Этот список

должен быть заключен в одинарные угловые скобки < >, а сами элементы списка – разделяться запятыми, например:

```
allocb <1,2,3,4>
```

При написании макроопределений иногда возникает необходимость в задании меток инструкций или имен полей данных. Поскольку каждое макроопределение может использоваться многократно, возникают ситуации, когда несколько имен или меток определены многократно, что вызовет ошибку трансляции. Для обеспечения правильной обработки таких ситуаций в макроязыке предусмотрена директива LOCAL, позволяющая локализовать заданные имена исключительно в данном макрорасширении. Формат:

```
LOCAL формальное_имя,...
```

Формальное_имя может затем использоваться в данном макроопределении с уверенностью, что при каждом макровызове его значение будет уникальным. В директиве LOCAL, если она присутствует, должно быть задано хотя бы одно имя, а если их несколько, они должны разделяться запятыми. Для обеспечения уникальности определенных директивой LOCAL имен ассемблер для каждого такого имени при каждом макровызове порождает реальное имя следующего вида:

```
??номер
```

Номер представляет собой 16-ричное число в пределах от 0000 до FFFF. Для предотвращения повторного определения имен программисту не рекомендуется определять в своей программе имена этого типа.

Директива LOCAL может использоваться только в макроопределении, причем, там она должна предшествовать всем другим предложениям макроопределения, т.е. следовать непосредственно после MACRO.

Для удаления макроопределений служит директива PURGE. Формат:

```
PURGE имя_макроопределения,...
```

Директивой PURGE удаляются все текущие макроопределения с указанными именами. Последующий вызов одного из этих макроопределений будет приводить к ошибке. Если имя-макроопределения представляет мнемонику инструкции или директивы, восстанавливается первоначальный смысл мнемоники в соответствии со значением данного ключевого слова. Директива PURGE часто используется для удаления ненужных макроопределений из подключаемой директивой INCLUDE библиотеки макроопределений. Библиотека макроопределений представляет собой обычный последовательный файл, который, в общем случае, может содержать большое число макроопределений. Комбинация директив INCLUDE и PURGE позволяет выбрать из них только нужные для данной программы, что сократит размер исходного файла.

Выход из текущего макроопределения до достижения директивы ENDM обеспечивается директивой EXITM, имеющей следующий формат:

```
EXITM
```

Выход из макроопределения по директивам ENDM и EXITM заключается в прекращении генерации текущего макрорасширения и возврате в точку вызова текущего макроопределения в динамически внешнем макрорасширении или в исходной программе. Ниже приведен пример законченного макроопределения:

```
add  MACRO     param
     IFB  param
     EXITM
     ENDIF
     ADD  AX,param
     ENDM
```

В этом мароопределении осуществляется добавление величины, определяемой формальным параметром param, к содержимому регистра AX. Блок условного ассемблирования IFB обеспечивает выход из макроопределния, если при вызове параметр не был задан.

15. Языки высокого уровня и Turbo Assembler

Ассемблер предоставляет программисту полную свободу действий при разработке программы, что одновременно является и его достоинством, и недостатком, так как требует от разработчика знания системы команд данного компьютера и его операционной системы. Кроме того, несмотря на минимальный размер выполняемого фрагмента при максимальной скорости работы, время, необходимое для создания программы, резко возрастает с увеличением объема разрабатываемого проекта. Поэтому ассемблер был и остается языком программирования для профессионалов.

В то же время программисты, работающие на языках высокого уровня, столкнувшись с ограничениями, которые неизбежны при использовании этих языков, не должны полностью переписывать свои программы на ассемблере. Чаще всего бывает достаточно перевода нескольких фрагментов кода, критичных по времени выполнения, чтобы все проблемы исчезли.

15.1. Вызов подпрограмм и передача параметров в языке C++

Язык Borland C++ поддерживает два способа передачи параметров в вызываемую функцию: стандартный способ языка C и с использованием модификатора pascal.

Рассмотрим программу на языке C++, использующую первый способ передачи параметров.

```
extern "C" void asm_f1(int *i1, int *i2, unsigned long l1);
void main(void)
{   int i=5, j=7;
     unsigned long l=0x12345678;
    asm_f1(&i, &j, l);
}
```

Строка

```
extern "C" void asm_f1(int *i1, int *i2, unsigned long l1);
```

определяет, что программа asm_f1 описана в другом файле. Особенностью компилятора языка C++ является то, что если не заданы специальные режимы, он добавляет перед своими идентификаторами символ подчеркивания (_). Таким образом идентификатор asm_f1 после компиляции будет выглядеть как _asm_f1. Это необходимо учесть при написании процедуры asm_f1 на языке ассемблера. Спецификатор "C" говорит о том, что используется интерфейс между программами, принятый в языке C. Строки

```
     int i=5, j=7;
     unsigned long l=0x12345678;
```

объявляют и инициализируют переменные i и j целого типа и переменную l беззнакового длинного целого типа. Далее вызывается функция asm_f1 на языке ассемблера и ей передаются три параметра: i, j и l. Передача параметров из вызывающей в вызываемую программу осуществляется через стек. В соответствии со способом, принятым в C , параметры передаются в стек справа налево, то есть сначала l, потом &j и далее &i. Следом за параметрами в стек помещается адрес возврата, то есть адрес команды в языке C++, которая следует за командой asm_f1(&i, &j, l);. Если для программы на языке C++ заданы малые модели памяти (tiny, small, compact), то таким адресом возврата является значение IP (для следующей после asm_f1 команды). Если для программы на языке C++ заданы большие модели памяти (medium, large, huge), то адресом возврата является значение CS:IP (для следующей после asm_f1 команды). Следует учесть, что для процессора 8086 данные в стек заносятся только словами (по 2 байта). Переменные i и j требуют двух байт памяти. Их адреса &i и &j (именно они заданы в программе) для малых моделей данных (tiny, small, medium) тоже требуют двух байт. Поэтому каждое значение &i и &j будет занесено в стек одной командой push МП 8086. Переменная l требует четырех байт памяти и заносится в стек двумя командами push МП 8086. Заметим, что малыми моделями памяти для кода являются tiny, small и compact, а для данных tiny, small и medium.Проанализируем выполняемый код, который будет построен компилятором языка C++ для этой

программы. Для этого опишем полученный выполняемый код на языке ассемблера.

```
push bp         ; сохранение регистра bp в стеке

; язык C++ сохраняет все локальные переменные в стеке (в нашем
; случае ими являются i, j и l). В результате после инструкций
; int i=5, j=7; unsigned long l=0x12345678; в область стека ;
; будут записаны следующие значения

mov bp, sp  ; запись в регистр bp адреса вершины стека
sub sp, 8   ; выделение в стеке места для локальных переменных
mov word ptr [bp-2], 5   ; сохранение в стеке значения i=5
mov word ptr [bp-4], 7   ; сохранение в стеке значения j=7
mov word ptr [bp-8], 5678h    ; сохранение в стеке младшей
                              ; части значения l=0x5678
mov word ptr [bp-6], 1234h    ; сохранение в стеке старшей
                              ; части l=0x1234

; после инструкции asm_f1(&i, &j, l); командами push в стек
; будут занесены передаваемые параметры. Это происходит в
; следующем фрагменте программы.

push word ptr [bp-6]; значение 0x1234 в стек
push word ptr [bp-8]; значение 0x5678 в стек
lea ax, [bp-4]      ; адрес [bp-4] в ax
push  ax            ; адрес [bp-4] в стек
lea ax, [bp-2]      ; адрес [bp-2] в ax
push  ax            ; адрес [bp-2] в стек
call _asm_f1        ; вызов функции _asm_f1
add sp,8            ; удаление из стека переданных параметров
mov sp,bp           ; удаление из стека локальных переменных
pop bp              ; восстановление регистра bp из стека
```

15.2. Вызов ассемблерных программ из программ на языке C++

При объединении программных модулей, написанных на языках C++ и ассмблера, эти модули должны использовать одну и ту же модель памяти. Кроме того для каждого модуля необходимо указать, какие переменные, объявленные в нем, передаются в другую программу и какие переменные модуль получает из другой программы. Чтобы обеспечить это, необходимо выполнить следующие требования:

1. Если процедура на языке ассемблера вызывается из программы на языке C++, то такая процедура в языке ассемблера должна быть описана как PUBLIC (должна появиться после дрективы PUBLIC). Аналогичное

описание необходимо и в том случае, когда две процедуры на ассемблере компилируются по отдельности.

2. Если необходимо обеспечить доступ к некоторым переменным из объединяемых программ, написанных на языках C++ и ассемблера, т.е. переменные должны быть глобальными, необходимо рассмотреть следующие случаи:
   - глобальная переменная объявлена в программе на языке ассемблера. В этом случае в программе на ассемблере она должна иметь атрибут PUBLIC, а в программе на C++ - extern;
   - глобальная переменная объявлена в программе на C++ как внешняя. Тогда в программе на ассемблере она должна иметь атрибут EXTRN;

Ниже приведен пример двух программ, первая из них написана на языке C++ и находится в файле C_ASM1.CPP, а вторая написана на языке ассемблера и находится в файле ASM1.ASM. Вторая программа вызывается из первой.

```
// вызывающая программа на языке C++
#include <stdio.h>
extern "C" void asm_f1(int*i1, int*i2, unsigned long l1);
void main(void)
{    int i=5, j=7;
     unsigned long l=0x12345678;
     printf("i=%d; j=%d; l=%lx\n", i, j, l);
     asm_f1(&i, &j, l);
     printf("i=%d; j=%d; l=%lx\n", i, j, l);
}
;  вызываемая программа на языке ассемблера
.model small
.code
PUBLIC C asm_f1
asm_f1 PROC near
     push bp
     mov bp,sp
     push si di
     mov si, [bp+4]
     mov di, [bp+6]
     mov bx, [bp+8]
     mov ax, [bp+10]
     mov cx, [si]
     xchg cx, [di]
     mov [si], cx
     pop di si
     pop bp
     ret
asm_f1  endp
     end
```

При вызове ассемблерной процедуры необходимо сохранять содержимое регистров, используемых процедурой. При возврате из процедуры значения регистров BP, SP, CS, DS и SS должны иметь те же значения, что до вызова процедуры.

Если вызванная ассемблерная процедура должна возвратить в вызывающую ее функцию некоторое значение, необходимо следовать правилам, которые учитывают особенность получения программой на языке C++ возвращаемых значений. Если процедура возвращает 16-битное значение, то оно должно быть помещено в регистр AX. Для языка C++ это значения данных типа char, short, int, enum и значения указателей типа near. Если процедура возвращает 32-битное значение, то оно должно быть помещено в регистры DX:AX (старшая часть в DX). Для языка C++ это значения данных типа long и значения указателей типа far.

15.3. Вызов программ на языке C++ из программ на языке ассемблера.

Как и выше, для каждой программы необходимо указать, какие переменные она передает в другой модуль и какие переменные она получает из другого модуля. Для этого необходимо выполнить следующее:

1. Если функция на языке C++ вызывается из программы на языке асемблера, то такая функция в языке ассемблера должна быть описана как EXTRN (должна появиться после дрективы EXTRN).
2. Описание всех глобальных переменных должно быть выполнено в соответствии с правилами, описанными в предыдущем параграфе.
3. Если функция на C++ возвращает значение, то оно будет помещено в регистры в соответствие с соглашениями языка C++.

15.4. Встроенный ассемблер (режим inline в программах на языке C++)

Наряду с рассмотренными возможностями Borland C++ позволяет аписывать ассемблерный код нпосредствнно в программе на языке C++. Любую ассемблерную команду можно записать в виде:

asm код_операции операнды ; или новая строка

Здесь «код_операции» задает команду языка ассемблера (например, mov), «операнды» – это операнды команды (например, ax, bx). Если с помощью одного слова asm необходимо задать много ассемблерных команд, то они заключаются в фигурные скобки. Комментарии можно записывать только в форме, принятой в языке C++. В программе на языке C++, использующей ассемблерные команды, необходимо задать директиву #pragma inline.

Рассмотрим пример.

```
#include <iostream.h>
#pragma inline
void main(void)
{    int a=10, b=20, c;
     cout<<"a="<<a<<"b="<<b<<'\n';
     asm mov ax,10  ; // в ax значение 10
     asm mul a      ; // умножение ax*a
     c=_AX;
     cout<<"c="<<c<<'\n';
     asm {
          mov ax, a
          mov bx, b
          xchg ax, bx
          mov a,ax
          mov b, bx
          }
     cout<< " a= "<< a <<"; b= "<< b <<'\n';
}
```

Литература

1. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке Ассемблер. Пер. с англ. – М.: Радио и связь. 1989. – 336 с.

2. Фролов А.В., Фролов Г.В. Библиотека системного программиста. Тома 18,19. MS-DOS для программиста, М.: Диалог-МИФИ, 1995. – 507 с.

3. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: "Диалог-МИФИ", 2001. – 288 с.

4. Финогенов К.Г. Самоучитель по системным функциям MS-DOS. – Изд.2. – М.: Радио и связь, Энтроп, 1995. – 382 с.

5. Сван Т. Освоение Turbo Assembler. – Киев: Диалектика, 1996. – 544 с.

6. Юров В., Хорошенко С. Assembler: учебный курс. – СПБ: Питер Ком, 1999. – 672 с.

7. Зубков С.В. Assembler. Для DOS, Windows и Unix. – 2-е изд. – М.: ДМК, 2000. – 640 с.

8. Скляров В.А. Программирование на языке ассемблера. Учебное пособие, М. Высш.школа, 1999, 152 с.

9. Юров В. Assembler: Специальный справочник. СПб: Питер, 2000.

10. Пирогов В. Ю. Assembler. Учебный курс. – М.: "Нолидж", 2001. – 848с.

11. Юров В. Assembler: практикум. – СПб: Питер, 2001. – 400 с.

12. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.

13. Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения. – СПб.: ООО "ДиаСофтЮП", 2002. – 656 с.

14. Ирвин Кип. Язык ассемблера для процессоров Intel, 3-е изд.: Пер. с англ. – М.: Изд. Дом "Вильямс", 2002. – 616 с.